Task-level programming for control systems using discrete control synthesis

Eric Rutten, Hervé Marchand

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Task-level programming for control systems using discrete control synthesis

Eric Rutten — Hervé Marchand

N° 4389

February 2002

THÈMES 4 et 1

ISSN 0249-6399 ISRN INRIA/RR--4389--FR+ENG

RHÔNE-ALPES

Task-level programming for control systems using discrete control synthesis *

Eric Rutten , Hervé Marchand

Thèmes 4 et 1 — Simulation et optimisation
de systèmes complexes — Réseaux et systèmes
Projets Bip et Vertecs

Rapport de recherche n° 4389 — February 2002 — 46 pages

Abstract: Robotic and control systems are ever more difficult to design, program, as well as to operate. This is due to their growing size and complexity. Therefore, their real-time programming and execution architectures require more user and design assistance, based on usable abstraction and tool support. Especially in safety-critical control systems (e.g., space robotics, robotic surgery) formal methods can be useful, but they have to be made usable by application domain specialists. User-friendly, domain-specific languages and interfaces ease the application of real-time analysis and implementation techniques. We are particularly interested in task-level programming of robot and control systems, where tasks encapsulate control laws. We consider the application of discrete control synthesis techniques, integrated into the framework of a task-level robot programming environment: they support the automated generation of correct controllers for control applications. We use the synchronous programming environment Signal, and the verification and synthesis tool Sigali. This requires to determine patterns of tasks and objectives, which are at once domain-specific to control systems, and generic enough to cover a broad class of control systems. We consider its application to safe discrete teleoperation. We illustrate this by a case study concerning the interactive discrete control of tasks of an excavating system.

Key-words: Control systems, real-time, safe design, discrete control synthesis, synchronous programming, robot programming.

* This work is partially supported by the IST project TELEDIMOS.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Programmation au niveau tâche des systèmes de contrôle-commande et utilisation de la synthèse de contrôleurs discrets

Résumé : La conception des systèmes robotiques et de contrôle-commande est de plus en plus difficile, ainsi que leur programmation et leur opération. Ceci est dû à leur taille et complexité grandissante. C'est pour celà que leurs architectures de programmation et exécution temps-réel requierrent plus d'assistance à l'utilisateur et au concepteur, foncée sur des abstractions utilisables et le support d'outils. Les méthodes formelles peuvent être utiles, particulièrement dans les systèmes de contrôle de sécurité critique (par ex., la robotique spatiale ou chirurgicale), mais elles doivent être rendues utilisables par des spécialistes du domaine d'application. Des langages et interfaces dédiés aux utilisateurs facilitent l'usage de techniques d'analyse et mise en œuvre temps-réel. Nous nous intéressons particulièrement à la programmation au niveau tâche en robotique et contrôle-commande, où les tâches encapsulent les lois de commande. Nous considérons l'application de techniques de synthèse de contôleurs discrets, intégrées dans le cadre d'un environnement de programmation au niveau tâche : elles permettent la génération automatique de contrôleurs corrects pour des applications de contrôle-commande. Nous utilisons l'environnement de programmation synchrone Signal, et l'outil de vérification et synthèse Sigali. Ceci requiert la définition de motifs de tâches et d'objectifs, qui soient à la fois spécifiques au domaine du contrôle-commande, et suffisemment génériques pour couvrir une large classe de systèmes de contrôle. Nous considérons une application à la téléopération discrète sûre. Nous illustrons ceci par une étude de cas concernant le contrôle interactif discret des tâches du système d'excavation.

Mots-clés : Systèmes de contrôle-commande, temps-réel, conception sûre, synthèse de contrôleurs discrets, programmation synchrone, programmation de robots.

1 Motivation

The general motivation of this work is to apply the relatively new techniques of discrete controller synthesis to the problem of designing safe control systems, especially in the area of robotics.

1.1 Safe programming of robot and control systems

Context. Our work is situated in the area of safe programming of critical applications, in the domain of control systems, where (sampled) continuous and discrete parts interact. These systems work embedded in an environment with its own dynamics, making them real-time systems as they have to adapt to its constraints. A typical occurrence of this is in the discrete event based sequencing of servo-control laws, requiring mechanisms for starting, interrupting, managing parallelism of the servoing tasks. In this domain, systems are often safety critical with regard to persons transported for example, or to hazards for the environment, or also because of the impossibility of a recovery intervention in case of remotely operated systems (space exploration, submarines, devices in hostile environments). Hence, the design of such controllers is particularly delicate, and it requires assistance tools and methods for:

- specification, through domain-specific languages useable by specialists of the application area rather than of real-time programming;
- validation, with automated verification tools, for properties for which there also is a need for domain-specific languages;
- and implementation, with automated code-generation, guaranteeing preservation of the semantics of the specification.

Approach. A variety of approaches to high-level robot programming exists [20] ranging from decisional autonomy and task-planning to component-based real-time programming. We are close to the approaches making use of mathematically defined programming languages, and their corresponding design environments based on formal models and methods. The advantage of such an approach is to inherit from a wealth of well-founded, efficiently implemented functionalities, regarding e.g., the specification and its testing or validation, the implementation with correct code generation, optimization.

Our work is related to the approach explored with ORCCAD (Open Robot Controller CAD) [1] [14]. It is the application to robot programming of reactive systems, synchronous programming and its associated technology. It proposes specialized languages for the specification of control laws, of Robot Tasks encapsulating them in a local discrete controller, and of Robot Procedures, sequencing Robot Tasks according to synchronizations between them or in reaction to sensor input. The discrete part of ORCCAD makes use of ESTEREL, one of the synchronous languages, and of its programming environment: compiler, simulator, and verification tool FC2TOOL: domain specific languages are compiled towards ESTEREL.

1.2 Discrete controller verification and synthesis

Validation and formal methods. The validation of such control systems involves different aspects, e.g. analysis of the robustness and stability of control laws, performance evaluation of the implementation code, quantitative scaling of the execution platform, verification of the real-time constraints and of the logical properties of the discrete part of the controller, like safety and reachability of states. These aspects are covered by a variety of validation techniques.

The verification of logical properties is supported by formal methods. Amongst them, a mature one is model checking [16], i.e. the computation of the satisfaction of temporal logics formulæ upon a (sometimes temporized, or hybrid) transition system or finite state automaton. The transition system models the possible behaviors of the controller, and the formulæ are in terms of the labels on the states and transitions (events, conditions) and quantifications over the paths taken along the transitions.

[1] Web site: `www.inrialpes.fr/iramr/pub/Orccad/`

Discrete controller synthesis. Discrete (or temporized, or hybrid) controller synthesis [59, 58] contributes to the safe design of controllers in that it automates their construction with formal model based techniques. It has the same basis as model-checking, but, instead of checking whether the model satisfies the properties, it consists of producing the transition system that has the sub-set of the behaviors of the previous one, restricted to those respecting the properties, by constraining the presence of controllable events. The resulting transition system is the most permissive one, in the sense that only the transitions harming the properties are inhibited, all the others being kept. This constructive feature seems very powerful and interesting for the automated generation of controllers, which are then correct by construction, in the sense that they are obtained as a result of a sound computation.

1.3 Discrete controller synthesis for task-level programming of control systems

Domain-specific application. In this context, the state of the art in discrete controller synthesis features models and algorithms for the computation of controllers, as well as tools implementing them, and a general methodology for their use. In the area of safe control systems programming that interests us, we would like to make use of these techniques, adapting them to our application domain: control systems, e.g., robotic systems, or more generally electro-mechanical systems. This comes within a wider context, where we want to make formal methods useable by domain-specialists, rather than formal-methods specialists. This is achieved by proposing domain-specific languages for tasks, missions, and properties. These specifications are in terms of the domain entities rather than in terms intrinsic to the underlying model. They are compiled into a model, built especially so that compilation, validation or synthesis can be performed.

The ORCCAD approach is an example of the integration of formal techniques into a programming environment [14], using the support of the synchronous technology [27]. The robot programming functionality based upon the use of discrete control synthesis can be seen at different levels. At the decisional level, it could be seen as a form of planning, deciding on sequencings leading from one situation to another. We are here more interested by its use for the partially automated sequencing/scheduling of tasks. The use of the resulting controller in an interactive framework is examined as a form of safe discrete telerobotics.

Points of focus. More precisely, we are particularly interested in:

- the specification: how to build a model of
  - the controlled process or device
  - partial controllers for sub-systems to be controlled
  - objectives to be satisfied (using temporal logics or observers)

  On the one hand, all of this has to be made manageable by a user who is not a specialist in this type of models and computation, on the basis on domain-specific languages and methodology. On the other hand, going from specification to the models must be done in a way favorable to the computation of the controller, e.g. in the choice and construction of controllable variables in the model.

- the execution of the resulting controller:
  - extraction of a deterministic controller in the set of solutions,
  - interactive or random choice of valuations (within the constraints) of controllable variables not determined by the environment.

  These are choices in the use of the computed controller, which has to be embedded in the control architecture of the system.

Our approach. In this report, we propose a systematic framework into which discrete control synthesis can be used in an automated way (i.e., user-friendly), on structures directly related to the domain of control system programming (i.e., robot control tasks and their sequencing) [61]. It comprises patterns for:

- task structures with phases and modes, transition events, events to be controlled by the synthesized controller;
- a mission structure, composition of a set of tasks, with maximal permissiveness;
- and objectives, pre-defined for user-friendliness, related to the task structure, and to the synthesis operations available.

This framework is intended to be generic enough to be automated into a programming environment component.

We also consider the interactive use of the synthesized controller as a form of discrete safe teleoperation, illustrated by examples inspired by excavation systems [65].

1.4 Organization of the paper

In the remainder of this paper, we will make a synthetic review of approaches in task-level programming of control systems in Section 2, in order to introduce the basic notions we are going to manipulate. Section 3 makes a review of the notion of discrete control synthesis, in a rather intuitive, user-oriented way (other exhaustive formal reviews are available elsewhere). In Section 4 we present with more detail the synchronous approach to reactive systems, and the techniques it proposes to support discrete control synthesis. We then describe a framework for using discrete control synthesis in control system programming in Section 5, featuring structures for behavior and objectives, and the methodology for generating and using the controller. An example inspired by excavation systems is treated in Section 7, illustrating the approach. Finally, Section 8 concludes, and lists a number of perspectives.

2 Task-level robotic and control system programming

2.1 The programming of discrete-continuous controllers

A variety of approaches exists in the domain of control systems and robotic architectures and programming environments. We consider robot control at task-level [20], where the continuous control part is encapsulated in a discrete, event-based control. This approach corresponds to a certain abstraction in the representation of robot behavior, the complexity of goals, the dynamicity of the environment, and the reactivity to its change.

Some works relate to artificial intelligence and make an emphasis on the autonomy of systems, notably with decisional aspects and particularly task planning [1] . Others are mainly focussing on real-time aspects at the operating system level, where the main difficulties are execution control and ensuring quantitative real-time properties. Control-Shell [20] is one such proposal. However, the abstraction level for design and analysis is then very close to the actual implementation.

Another point of emphasis is the *safety-critical* nature of robotic systems, and the need for validating them. This means that the functionalities expected from a programming environment involve the presence of a *formal model* (e.g., a finite state automaton or transition system), and its exploitation as a basis for automated analysis, validation and implementation.

An example is the ORCCAD approach [14]: we will detail some of its features further in order to introduce our subsequent choices of modelling. It combines practical use of a reactive language and associated formal tools for specification and analysis, with code generation producing real-time executives. Yet another example of using a formally based, practical programming environment for the specification, verification and implementation of a robot system has concerned the synchronous language SIGNAL [45]. It supported the combination of continuous control laws described in a data-flow style, and the sequencing of tasks using a task structure. The verification of a number of safety properties could be performed, using the SIGALI tool associated to SIGNAL; they concerned e.g., the correct sharing of the control of degrees of freedom by the control laws.

Production systems and manufacturing machines have also been an area where formal methods have been used to model possible behaviors and analyse or control them, the most used model being Petri nets; a variety of works exists on their supervisory control [19, 2, 32].

2.2 The Orccad approach

We give here some details about the ORCCAD robot programming environment [14], as its concepts are a basic influence to our work. We are interested specifically by:

- the nature of the entities proposed for specification,
- their formal definition, by way of an encoding into a synchronous language with a mathematical semantics,
- its use for compilation, analysis and verification

2.2.1 Specification in Orccad

The entities proposed to the designer correspond to the domain-specific objects naturally manipulated in the application area.

Modules and control laws. The kernel component defining the behavior of a robot is a control law, defining the servoing relation between sensors and actuators. Such a control law is a function decomposed into more or less complex operations. Modules can be defined to implement each a computation (e.g., filters, matrix operations, Jacobian inverse). A data flow network of such modules can be assembled, through a graphical interface, with data exchange and synchronizations between modules. Such a data flow network implements a control law.

Robot tasks. Tasks in ORCCAD are defining an encapsulation of a control law, i.e. a data flow network of modules, into a local discrete controller, with basic events management, where starting, completing or interrupting the task is taken care of. The latter includes an exception mechanism, where events can have either a local effect (change of parameters), or interrupt the task (and lead to an error-treatment task), or a fatal effect, bringing the whole mission to a safe stop. These tasks also have an initialization phase, where internal state (e.g., filters) are computed, but the control can not yet be emitted. The basic discrete behavior of these tasks is modeled as an *automaton*, encoded in the synchronous language ESTEREL [12, 27], hence making it possible to build a formal model exploitable by the synchronous tools (compilation or verification, see further).

Robot procedures. Missions are built up by assembling tasks into an imperative control structure, with various sequential, parallel or conditional control structures, defining a fully-determined sequencing. It can be noted that given the exception handling and initialization phase mentioned above, the synchronizations between tasks is not trivial, and their coding into ESTEREL not easy. A contribution to a domain-specific environment for programming such missions is the language MAESTRO [69].

The resulting model obtained through the ESTEREL encoding of a robot-procedure and of the robot-tasks is the *global automaton* of the application. States correspond to activation status of different modes of the different tasks, transitions correspond to changes thereof, and labels associated with transitions carry the events causing, conditioning or signalling these changes.

2.2.2 Analysis and verification in Orccad

Using formal models and tools in done in an integrated way, where, from the *domain-specific interface* mentioned above, a transition system is automatically constructed through the use of ESTEREL compilation. Therefore it concerns the discrete-event part of the controller. This way, the synchronous techniques are encapsulated into the programming environment, i.e., the interface is domain-specific and user-friendly, the synchronous model is built in order to be verifiable by the tools in an automated way. The automata can be observed, or, when too complex, transformed using the abstraction criteria so as to have only the part of the behavior that concerns the property under consideration. The tool used for this is FC2TOOLS, related to ESTEREL [27, 28]. Another formal analysis also related to ORCCAD is the use of (max,+) algebra techniques for the validation of multi-tasks systems with the taking into account of temporal performances [8]. The

global, general approach is to first design a system or controller, by specifying its correct behaviours, and taking care to manage all possibly problematic situations. After this, there is a phase of validation using different techniques, with a diagnostic possibly leading to re-design.

By comparison, in this paper, we explore a rather different design philosophy, closer to control theory, where first a model is built of all possible behaviours, then characteristics defining the correct ones are specified, and finally an algorithm is applied, computing the restriction to the correct behaviours, which constitutes the controlled system; the restricting constraint itself is the synthesized controller.

3 Discrete control synthesis

This section briefly and intuitively introduces some basic notions of discrete control synthesis. It focuses on those used in the remainder of the paper; fundamental presentations, out of the scope of our work, are available elsewhere [59, 18], as well as comparative and illustrative studies (e.g., [58][51]).

Further in this paper, details will be given on the synchronous approach to reactive systems, particularly related to our context, and how control synthesis is treated there.

3.1 What it is, intuitively and informally

In the fields of manufacturing systems, robotics, etc, many applications require high reliability and safety. Traditionally, these requirements are checked *a posteriori* using simulation techniques and/or property verification [16]. Control theory of discrete event systems allows the use of constructive methods that ensure, *a priori*, required properties on the system behavior. In this approach, the validation phase is reduced to properties not guaranteed by the programming process. There exist different theories for control of Discrete Event Systems since the 80's [59, 9, 30, 42]. Usually, the starting point of these theories is: given a model for the system and the control objectives, a controller must be derived by various means such that the resulting behavior of the closed-loop system meets the control objectives.

3.1.1 Behaviors as transition systems

The basic models are discrete-event systems, and can be formulated as, e.g., formal languages [59], Petri nets [30], or finite state machines, also called transition systems [18]. In the latter case, states correspond to given configurations of e.g. the robot system, and/or of the activation status of tasks controlling it. Transitions correspond to the occurrence of, e.g., events, commands, thresholds modeling the possible actions that the system could perform. They are labels on the transitions, which can involve conditions on their fireability. At each instant not all but only a part of the actions are admissible depending on the state the system evolves in. This is modeled by a transition relation which gives access to the "next states" according to the current state of the system and the event that occurs. The behavior of the system is simply given by a sequence (or a path) finite or infinite of transitions accepted by the system. When modeling realistic systems, it is often convenient to manipulate state/event variables instead of simply atomic states/events. Within this framework, the states (as well as the events) can be seen as a particular instantiation of the vector of variables. Most of the times, these variables take their value in a finite set, leading to a finite state space. We will come back to this aspect further in the paper when dealing with SIGNAL specification and symbolic controller synthesis using the tool SIGALI.

Figure 1: The behaviours of a 1-bit counter, and its input-ouput profile.

For example, Figure 1 illustrates a transition system behaving like a 1-bit counter: it has two states, corresponding to the values 0 and 1, and can be either incremented (i) or doubled (d), or both together (di), in which case it behaves as doubling and then incrementing. Transitions exiting state 1 have a label featuring a carry (c), meaning that this event is present when these transitions are taken. When seen as a process with inputs and outputs, this behavior has a profile as shown in Figure 1, with input events i and d, and outputs s (state value) and c (carry event).

Modelling interaction by composing automata. In many applications and control problems, Finite State Machine (FSMs) are the starting point to model fragments of a system, which usually consists of many different sub-systems that are further composed according to some operations that make the sub-systems cooperate through either states or events. There exist various kinds of composition like synchronous composition, interleaving, etc (see [18, 16] for more details). Synchronization can be performed either on states or on labels of the transition (i.e. events). When dealing with systems modeled by means of variables, it is also possible to let components share common variables.

We consider synchronous composition in the way adopted by the synchronous model and languages [27], i.e., intuitively, in a given global state, every local transition system that can make a transition upon its inputs, does so.

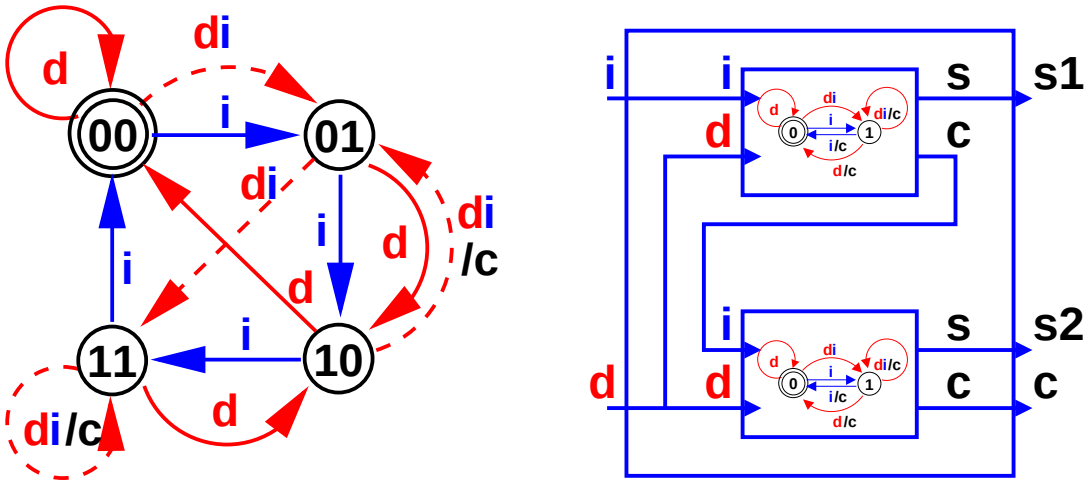

Figure 2: The behaviours of a 2-bit counter, and its input-ouput profile.

For example, Figure 2 illustrates how a 2-bit counter can be constructed by composing two 1-bit counters, behaving like the transition system shown. The global process has the shown interface, with input events i and d, and outputs s_1 and s_2 (state values) and c (global carry). It is constituted by the composition of two instances of the 1-bit counter of Figure 1. A first one is taking i as increment input, and d as doubling input, and outputting s_1 and a local carry. The latter is given as increment input to the second instance of 1-bit counter: this means that the c output of the one is equal to the i input of the other. The same d as before is used as doubling input. The outputs are the state value s_2 and the 2-bit counter carry. The automaton shown in Figure 2 gives the global automaton resulting from the composition, which describes the behaviors of the 2-bit counter: states are defined by the values of $s_1 s_2$, and transitions are labelled by the presence of events d, i, and c.

This set of behaviors can be constrained by considering synchronizations of the inputs. E.g., it can be a fact known about the environment of the counter that never do i and d occur together. In this case a reduced automaton describes the resulting behaviors, as shown in Figure 3. This exclusivity concerns the profile of the 2-bit counter; it does not prevent us to make use of the simultaneity of local, internal d and i inputs, in order to take care of the presence of a carry, and forwarding it from lower weights to higher weights when doubling. Constraints of that kind exist, other than exclusivity, like inputs with constant values or presence/absence, or inclusion or implication relations, as when the presence of one input implies that of

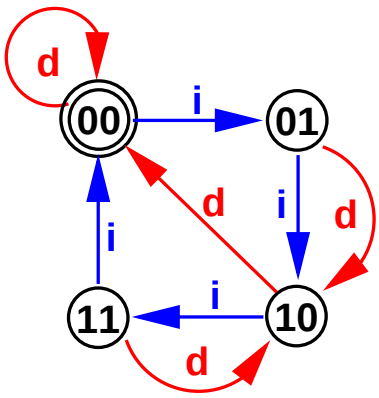

Figure 3: The behaviours of a 2-bit counter, with exclusive inputs.

another one (i.e., the latter includes the former). The assertion of such constraints reduces the automaton by removal of transitions carrying contradicting labels, and hence possibly of states reachable only through these transitions.

3.1.2 Properties and objectives

Such transition systems can have properties related to the reachability of some subset of the state space, or to the existence of paths along which a certain sequence of events exists. Examples of relevant practical properties are:

- correct use of resource (e.g., mutual exclusion),
- correct event ordering and precedence (e.g., w.r.t. consistency of data transformations),
- desirable dynamical behavior (e.g., absence of deadlock or livelock),
- coordination towards a goal (e.g., necessity of the achievement of a given situation)

Checking these properties upon a given automaton is the functionality offered by model-checking [16].

For example, an interesting property can be that some Boolean expression upon state variables and events always has the value *true*, at all states reachable from some initial situation. If it is true, the transition system is said to be *invariant* w.r.t. this property. In other words, from any state in that subset of the state space, *all* transitions remain in that subset. There is a notion of *greatest invariant subset* of a state set.

Another interesting property is that there exists a reachable subset of the state space where the expression is true; i.e., there is a subset such that at each state at least one transition can be taken staying in the subset. If this property holds, the transition system is said to be *invariant under control* w.r.t. this property. In other words, from any state in that subset of the state space, *some* transition remains in that subset. Here too, there is a notion of *greatest invariant under control*. Interestingly, w.r.t. to our goals of control synthesis, this means that there exists a subset where the property holds, and that it may be possible to constrain the behavior to stay there.

Differently to these safety properties, another interesting issue is the achievement of tasks, involving reachability/attractivity of states (e.g. the marked states of the systems, or states (not) having a desired property) through event trajectories. Informally a set of states is said to be *reachable* from the initial situation whenever, from all states that can be reached from the initial states, there is *some* trajectory that goes to the considered set of states. In terms of the previous notions, a set of states is reachable if it is not included in the greatest invariant of the complement of the initials in the state space.

Similarly, a set of states is *attractive* whenever *all* the possible trajectories of the system lead to the considered set of states. In terms of the previous notions, a set of states is attractive if it is not included in the greatest invariant under control of the complement of the initials in the state space.

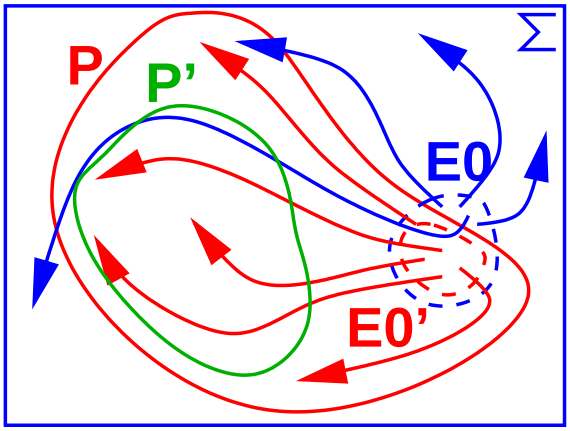

Figure 4: Properties of a subset E of the state space Σ

An example illustrating the previous points is shown in Figure 4. There, a subset of the state space Σ with some property P is *not invariant* if *E0* is initial but is *invariant* if *E0'* is initial, whereas a subset with some property P' is *reachable* from *E'0*, but *not attractive* from *E0'*, because there exist some trajectories that do not reach *P'*.

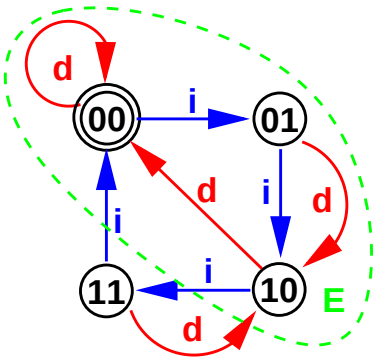

Figure 5: Properties of E in the behaviours of a 2-bit counter.

Coming back to the 2-bit counter example, let us consider the set of states E with the state property `not (X1 and X2)`. It comprises the states 00, 01 and 10. It is easy to check, as shown in Figure 5, that E is reachable from 00 (i.e., *from the initial state*), but is *not* invariant (i.e., *some control is necessary*). However, it is invariant under control (i.e., *the control is possible*).

3.1.3 Controllable or uncontrollable events

Related to the previous point, it is useful to be able to distinguish between the events perceived as *uncontrollable* (typically inputs received from sensors, exceptions or priority events, failures, tick of a clock), from those upon which a decision can be taken by the discrete controller (typically the starting of some task), called *controllable*.

The notion of controllability here is not exactly the same as in the notion of invariance under control introduced above. The latter means the *existence* of a transition (whatever its labelling events are) remaining in the appropriate set. Whereas here we mean that some events can be constrained in order to enforce the control.

However the notions are related. What we would like to have is of course that the constraints that make the property hold do not concern the uncontrollable inputs, but rather that constraining controllable events

is sufficient to keep the behavior inside the proper subset of the state space. More generally, synthesis will consist of exploring how to constrain the controllable events in order to achieve the objective.

3.1.4 Synthesis

The synthesis of a controller consists of automatically computing the *controller*: a relation that, given a state and uncontrollable events, gives the value of controlled events such that only transitions respecting the objectives can be taken (in other words: contradicting behaviors are inhibited), as illustrated in Figure 6 in the case of a deterministic controller (i.e., a function $\Sigma \times Unctrl \rightarrow Ctrl$).

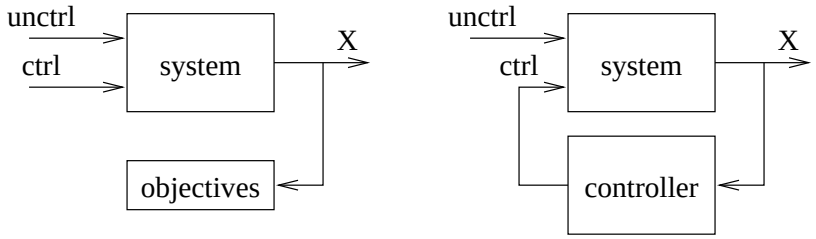

Figure 6: Discrete control synthesis: from uncontrolled system (left) to closed-loop (right).

This produces a constrained model, i.e. model and controller together satisfy the property[2]. It involves restrictions on admissible initial states, as well as on transitions. In the framework depicted in Figure 6, the control strategy is the following: given a state and a set of uncontrollable events that occurs, the set of controllable events that may occur is given by the controller according to the restrictions on transitions computed during the synthesis phase. There can be several controllers satisfying the property; actually, sometimes forbidding any move is a control which avoids the states not satisfying the property, but this is less than satisfactory w.r.t. the activity of the control system. The notion of maximally permissive controller captures that we have the controller which insures the properties satisfaction while keeping the greatest subset of behaviors of the original, uncontrolled, system.

Considering our 2-bit counter example, first, assume that the two events `i` and `d` are controllable, the objective is then to *make* `not (X1 and X2)` *invariant*. It is easy to see that to solve this problem it is sufficient for the controller to *inhibit event i from state 10*. This is illustrated by Figure 7.

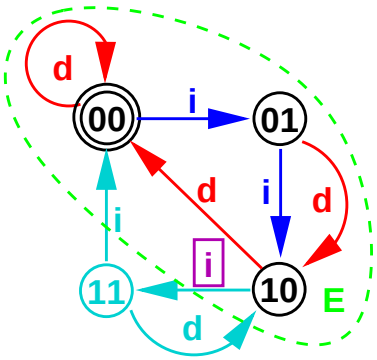

Figure 7: Controlled behaviours of a 2-bit counter.

Further, if we analyze the result one can see that `i` needs to be controllable, that is to say if `i` had been declared uncontrollable, then there would exist no solution to the control problem. On the other hand, *d* does need to be controllable actually. In this case, the result would have been the same. Among other, this small example raises the problem of "what needs to be controllable" and "does there exist a notion of minimality in terms of number of controllable events".

[2] Comparatively, verification, given a model and property, produces a satisfaction diagnostic.

Alternately, this small example of an automaton can be re-interpreted, in a slight variant, as a two-task control system, as illustrated by Figure 8. The state describes activity of tasks (i.e. active whenever Xi is act*i*, and inactive otherwise: idl*i*). The initial state is that both tasks are idle. We have labels start*i* and stop*i* which start and stop the tasks according to the cases. The transitions describe the behaviours of the particular system (not all combinations are admitted). The previous control problem would be here to insure that the two tasks are never simultaneously active. Concretely, the solution is to *forbid to start task 2 when task 1 is active.*

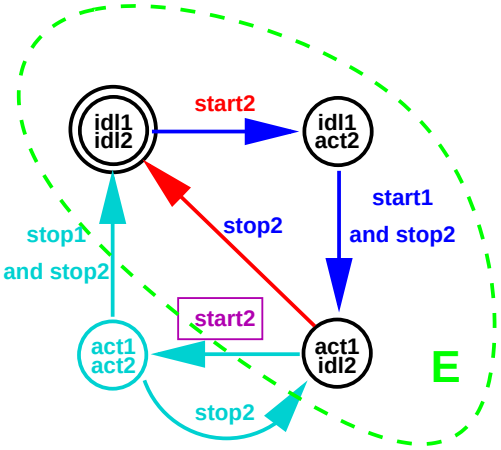

Figure 8: Controlled behaviours of a 2-tasks system.

3.1.5 What to do with the synthesized controller?

Discrete control synthesis is a constructive manipulation, in the sense that it synthesizes a controller, i.e. a program which decides, upon an input event and a current state, what a possible controllable event is such that the transition is compatible with the properties. This controller can be put to work, in different ways, such as simulation of the controlled system (controller in parallel with the uncontrolled system), interpretation of the control function step by step, or execution of a more compiled form.

In case of non-determinism, i.e. given a state and an (uncontrollable) input, there exists a set of possible controls, a choice must be made in order to obtain a controller which is a function of state and input. At this point various strategies can be chosen:

- *Arbitrary (random) choice* among the set;
- *"Optimal" choice* according to a quantitative criteria over either states or events (evaluated at run-time, as a difference with optimal synthesis [64, 46] where the controller produces an optimal control, as a result of an off-line analysis and evaluation);
- *Interactive choice* through a menu or panel. In a robotics application perspective, this can be interpreted as discrete teleoperation.
- other solutions can be explored, with some characterization of the sequences, like having series with always different values.

3.2 Models and methodology

There exists a background of theoretical results on the subject, of which we make no exhaustive study, but we give references to approaches also other than the one we will adopt.

The basic models are discrete-event systems, and can be formulated in terms of formal languages [59], Petri nets [2, 32, 33], dynamical equations systems [47] or finite state machines (possibly hybrid [41, 7]).

The latter cases take us close to the models that are at the basis of synchronous techniques. The transitions between states are labelled, typically with events: in reactive systems, the transitions can be taken upon the occurrence of these events. The states can be characterized by valuations of variables.

3.2.1 Models of Discrete Event Dynamical Systems

The works of Ramadge and Wonham are a basic reference [59, 18, 40]. Developments have seen a diversification of the formalisms used, all meant to describe discrete event dynamical systems (DEDS) [25], including among others, language based specification [59, 18], Petri nets [2, 32], real-time temporal logics, net condition/event systems, "exotic" algebras, e.g. (max,+) (which also have seen application to robotics programming [8]), dynamical equational systems [47], finite transition systems [59]. There exist comparative studies between them, some illustrated by examples [58]. The two latter are of equivalent expressiveness, related to languages theory, and have extensions such as temporized and hybrid systems.

The basic notions, illustrated above in Section 3.1, have been formally introduced in terms of languages theory [59, 40]:

- *controllable transitions system*, meant to model the plant to be controlled, where some transitions can be inhibited according to controllable events (the other events being uncontrollable); it comprises all the possible behavior of the plant including the "undesirable" ones. Its behavior has to be reduced according to the *specification* which describes the admissible and expected behaviors of the plant.
- *controller* or *supervisor*: the function defining for each state the set of authorized controllable events; it plays the role of observer and feedback function; This function can be extended to a trajectory of the plant; i.e. given a trajectory taken by the plant so far, the goal of the controller is to send back to the plant the set of authorized controllable events (i.e. the choices made by the controller do not only depend on the current state of the plant).
- *closed-loop behavior*, of the composition of controller and controlled system;
- *controllability*: the existence of behaviors of the controlled system respecting the controller, accepting all uncontrollable events,
- *non-blocking*: all the behaviors have the opportunity to end in a marked state of the plant.
- *most permissive controller* in the sense that there does not exist a controller or supervisor such that the closed-loop behavior of this controlled system is "greater" (w.r.t. language inclusion) than the previous one.

Extensions of these notions concern modularity and distribution (decentralized control) [40], control under partial observation and optimization criteria [64] (among all the admissible controllers some are better than others according to a quantitative criterion or order relation), quantitative time.

There have been developments of practical software tools, relying on algorithms similar to those used for model-checking purposes, which have reached a notable maturity concerning scalability. The algorithms are various, some perform a backward search, from the goal set of states, back to the set of states and transitions following which the goal can be reached, using a fix point computation; others use more heuristic-driven techniques (see Section 3.2.4). Some of the techniques have been confronted to applications of a size that was naturally challenging in terms of complexity, e.g. a robotic production cell [19, 15], and the incremental specification of an electrical power transformation plant [49] using the Signal programming environment, using a methodology which we will present further.

3.2.2 Timed and hybrid automata

Temporized systems extend finite transition systems by associating clocks to states, and allowing transitions upon thresholds, and resetting operations. The tool KRONOS implements the model-checking for such temporized systems. It has been used in the verification of temporal properties of robot applications in relation with ORCCAD [34].

For these formalisms, an approach to controller synthesis has been described as finding a winning strategy in a game between controller and environment, where the outcome depends on the player's actions as well as their timing [41]. The use of KRONOS for controller synthesis is based on this work, with a fix-point operation [3]. Another technique is an on-the-fly method [68], where a forward search returns the first solution. Two algorithms are proposed: one for invariance, the other one for reachability; they are adapted to timed automata. This method has been applied to the modeling and synthesis of real-time non-preemptive schedulers [5, 4]. Petri nets with synchronization modes (AND, MAX, MIN) are used for the specification of the system and the environment; they are translated to timed automata with deadlines, which are given, together with a property (e.g. exclusivity on a shared resource), to the synthesis algorithm. It produces a timed automaton representing correct schedules (as a sequence with dates). Among the examples treated, there is a robotic arm controller. Another application is in a multi-media documents management system with temporal constraints [22].

Another approach in timed models, using Metric Temporal Logic to express requirements, characterizes itself by its relation with planning in artificial intelligence, and the forward-chaining nature of its search algorithm [11], not considering the computation of supremal solutions (i.e., complete, most permissive). It is supported by a tool: TCST (Temporal Controller Synthesis Tool), and has been applied to cases studies [10].

Linear hybrid systems have discrete states associated with linear differential equations [7]. The backward search (through a predecessor function) of the maximal sub-system respecting a property involves problems in effectiveness and convergence. Approximation techniques are used to define an over-approximated successor function, and an under-approximation of the predecessor function, leading to an effective algorithm. A tool called D/DT implements algorithms for verification and synthesis.

3.2.3 Supervised control, PLCs and Grafcet

In the previous approaches to supervisory control theory, the supervisor can only prevent some events from occurring, hence preventing the process from taking corresponding transitions, but it can not force them. However in a practical setting, where the execution of the controller manages tasks (starting and stopping them), orders have to be emitted outwards, and outputs have to be distinguished. The supervised control approach separates control (emitting forced events to the process) and supervision (preventing the controller from emitting forced events), in a hierarchical framework [19]. This involves the design of a process controller, with emissions of controllable events; in order to be controlled by the supervisor, their emission has to be conditioned by authorizations from it. This work integrates the use of Grafcet for specification and implementation on programmable logic controllers. There are other approaches related to PLC programming [53].

3.2.4 Algorithms and tools

Considering Finite State Machines (FSM) as model of the system we want to control, there exist several tools allowing controller synthesis, other than those mentioned above; among them:

- Based on enumerative techniques, UMDES-LIB [70] as well as CTCT [71] are tools allowing the study of discrete event systems modeled by FSMs. Both allow the manipulation of FSMs and have classical operations of supervisory control theory. UMDES-LIB is developed by the DEs group of the University of Michigan and CTCT[3] by the university of Toronto.

- Based on symbolic techniques, relying on a Decision diagram implementation, there is also STCT [72] developed at the university of Toronto, and CSLXT [51] (Siemens) (see also [26]).

Among the available methods and tools, SIGALI is integrated with the SIGNAL environment for the design of real-time systems. It forms a complete toolset for undertaking experiments, as it provides for the quite rare combination of:

- a high-level specification language, SIGNAL, facilitating the description of behaviors in a structured way, with possibilities for modularity and reuseability;

[3]Available at http://www.control.utoronto.ca/DES/

- a complete compilation environment, with optimization, distribution and code generation for various execution or simulation platforms;
- a formal computation tool performing symbolic verification and discrete control synthesis [47, 50].

With this all, it is possible to experiment with models, properties, and perform interactive simulations of the obtained controllers.

3.3 Applications of discrete controller synthesis in control systems and robotics

There are works on fundamental issues related to automated manufacturing regarding e.g., resource managing, boundedness, reversibility [2]. Some studies have been experimented with on particular manufacturing systems [19]. Closer to our approach, i.e., integrating synthesis into a programming process, model-checking can be used within a heuristics-guided planning algorithm [52]. Differently to these works, we do not try to propose new algorithms or models, but we want to apply existing ones. Our goal is to define a framework, not specific to an application, but generic enough for a wide range of robotic systems, in other words, the framework should be *domain* specific but not *application* specific. This genericity serves as a foundation for a design process where a tool support is integrated into the compilation from high-level specification of tasks and mission (in terms of the domain) to the actual, implemented discrete controller.

Hence we need tools, and as explained before, we are particularly interested in results related to the synchronous approach. In this area, a particularly mature tool is SIGALI, integrated in the SIGNAL environment (see `www.irisa.fr/espresso`).

4 Sigali, Signal, and the synchronous approach to reactive systems

4.1 The synchronous approach to reactive systems

The synchronous approach to real-time programming [27] and the languages and environments related to it are characterized by a formal foundation of the languages, where a notion of synchronous composition allows to consider synchronizations of processes at specification level, independently of their implementation. Based on the formal models, the environments offer a set of tools for analyzing (e.g., verification [48], performance evaluation [35]), transforming (e.g., optimizing, pruning of dead code) or implementing (e.g., generating code for different execution platforms) the programs. The tools forming the synchronous technology have a certain interoperability through the use of exchange formats like DC [13].

In this paper, we will be focusing on SIGNAL and SIGALI, as they offer an appropriate set of tools for our purposes. The other approaches are however related in various ways to our interests. ESTEREL is an imperative language, specially well-suited for the description of event-based sequencings. It is used in ORCCAD as the output language, encoding the model of tasks and procedures [14]. It provides the support for the use of verification tools, as well as code generation, related to ESTEREL. LUSTRE is a data-flow language [29] different from SIGNAL, but which could have been used too for the description of behaviors to be presented in further Sections. MODE AUTOMATA have been introduced in order to facilitate the specification of switching modes, each defined by a data-flow process in LUSTRE [44]; this way it combines advantages of ESTEREL and LUSTRE. All these languages can be compiled into common exchange formats, produced as output and accepted as input by the different synchronous analysis and implementation tools.

4.2 The Signal language and environment

Based on the synchronous paradigm, the SIGNAL language has been developed, with associated set of tools: the academic SIGNAL/SIGALI toolset (graphical editor, compiler, code generator, model checker, simulator), and the commercial SILDEX tool[4]. Before giving formally the semantics of SIGNAL let us just say that a SIGNAL program describes relations between flows of data and events. The compiler transforms the program into a system of equations and then calculates the solutions of the system. The compilation of SIGNAL code provides a dependence graph on which static correctness proofs can be derived: it automatically checks the

[4] `www.tni.fr/frame-sommaire.eng.html`

network of dependencies between data flows, and detects causal cycles, temporal inconsistencies from the point of view of time indexes. SIGNAL automatically synthesizes the scheduling of operation involved inside a control loop (note that this work is often an error-prone task when done by hand in classical C-like language), and this scheduling is proved to be correct from the point of view of data dependencies. Further, the compiler synthesizes automatically *global optimizations* of the dependence graph, following different criteria. Then, according to some formal transformations of the graph, the user can choose to generate either *Embedded code* or code dedicated to simulation, or performance evaluation. At the same time, SIGALI, the model checker (also used for controller synthesis purposes) allows us to prove dynamical properties. All these functionalities are integrated in the SIGNAL environment, which is organized around the hierarchical synchronized data-flow graph. In that sense, the whole design process requires no manual transformation of models from one tool to another. SIGNAL can then be seen as a fully integrated environment.

The SIGNAL language allows to specify multiclock dynamical systems following a block-diagram style. Blocks represent dynamical systems, and can be connected together, to form higher blocks, and so on. SIGNAL programs involve *signals* and relate them together via operators. Signals are typed sequences (boolean, integer, real,...) whose domain is augmented with the special symbol $\perp$ to denote absence. In SIGNAL, symbol $\perp$ is not handled explicitly by the user, this prevents the user from manipulating explicitly time indices. SIGNAL has a small number of primitive constructs, listed [5] in Table 1.

Language Construct	Signal syntax	Description
stepwise extensions	C := A op B	where op : arithmetic/relational/boolean operator
delay	ZX := X $ n	memorization of the n^{th} past value of X
extraction	C := A when B	C equal to A when B is present and true
priority merging	C := A default B	if A is present C:=A else if B present C:= B else C absent
Process Composition	(|P|Q|)	processes are composed, common names correspond to shared signals
useful extensions		
	when B	the clock of the true instants of B
	event B	the presence instants of B
	A^= B	Clock of A equal with clock of B

Table 1: Basic SIGNAL language constructs

4.3 Sigali, verification and synthesis

The SIGNAL programming environment[6] [39] and its associated verification and synthesis tool SIGALI [48], where the controller synthesis methodology is integrated from specification to simulation of the synthesized controller [47]. We here restrict ourselves to SIGNAL processes involving only Booleans and *clocks* (i.e., pure signals with domain $\{true, \perp\}$). Such SIGNAL programs are equivalent to arbitrary finite state machines. Controller synthesis is conveniently performed by embedding SIGNAL into dynamical systems over the finite field $\mathbb{Z}/3\mathbb{Z} = \{-1, 0, +1\}$ with special rules $1 + 1 = -1$, and $-1 - 1 = +1$. The following coding is used $true \mapsto +1$, $false \mapsto -1$, $\perp \mapsto 0$. Using this coding, SIGNAL primitives translate as shown in Table 2.

In this table, x, x' denote auxiliary current and next state variables, these are needed to encode the delay operator. Any SIGNAL specification can then be translated into a set of equations called polynomial dynamical system (PDS) of the form :

$$S \quad = \quad \left\{ \begin{array}{lcl} X' & = & P(X, Y, U) \\ 0 & = & Q(X, Y, U) \\ 0 & = & Q_0(X) \end{array} \right. \tag{1}$$

where X, Y, U, X' are vectors of variables in $\mathbb{Z}/3\mathbb{Z}$ and $dim(X) = dim(X') = n$. The components of the vectors X and X' represent the current and next states of the system and are called *state variables.* They originate from the translation of the delay operator. Y is a vector of variables in $\mathbb{Z}/3\mathbb{Z}$, called *uncontrollable event variables*, whereas U is a vector of *controllable event variables.* The first equation is the *state transition*

[5] We list only primitive statements. The actual syntax involves derived operators e.g., for handling constraints on clocks.
[6] Web site: `www.irisa.fr/espresso`

B := not A	$b = -a$
C := A and B	$c = ab(ab - a - b - 1)$ $a^2 = b^2$
C := A or B	$c = ab(1 - a - b - ab)$ $a^2 = b^2$
B := A $1 (init b_0)	$x' = a + (1 - a^2)x$ $b = a^2 x$ $x_0 = \mathtt{b}_0$
C := A when B	$c = a(-b - b^2)$
C := A default B	$c = a + (1 - a^2)b$

Table 2: Equational encoding of SIGNAL language constructs.

equation; the second equation is called the *constraint equation* and specifies which event may occur in a given state; the last equation gives the initial states. The behavior of such a PDS is the following: at each instant t, given a state x_t and an admissible y_t, we can choose some u_t which is admissible, i.e. such that $Q(x_t, y_t, u_t) = 0$. In this case, the system evolves into state $x_{t+1} = P(x_t, y_t, u_t)$.

Verification Algebraic manipulations allow for the definition of operations checking satisfaction of properties like:

- invariance of a transition system w.r.t. a condition (also called security) where from every state satisfying it every transition leads to a state also satisfying it;
- control-invariance of a transition system w.r.t. a condition where from every state satisfying it *some* transition leads to a state also satisfying it;
- reachability of a set of states from another one, where some path initiated in the latter reaches the former;
- attractivity of a set of states from another one, where *all* paths initiated in the latter reach the former.

It is also possible to symbolically express CTL formulae [21], propositional μcalculus formulae [36, 56] as well as bisimulation equivalences [57]. For a more complete review of the theoretical foundation of this approach, the reader may refer to [37, 38]. Examples of verification can be found in [6, 48].

Synthesis. These notions can be used in control synthesis, where a transition system can modified by constraint on events declared controllable, *making* it satisfy a property [47]. A transition system can be submitted to a series of such operations, in a process of incremental synthesis. Given a PDS S, as defined by (1) a controller is defined by a system of two equations $C(X, Y, U) = 0$ and $C_0(X) = 0$, where the latter equation $C_0(X) = 0$ determines initial states satisfying the control objectives and the former describes how to choose the instantaneous controls; when the controlled system is in state x, and an event y occurs, any value u such that $Q(x, y, u) = 0$ and $C(x, y, u) = 0$ can be chosen. The behavior of the system S composed with the controller is then modeled by:

$$S_c = \left\{ \begin{array}{rcl} X' & = & P(X,Y,U) \\ 0 & = & Q(X,Y,U) \;=\; C(X,Y,U) \\ 0 & = & Q_0(X) \;=\; C_0(X) \end{array} \right. \tag{2}$$

Using algebraic methods, avoiding state space enumeration, we can compute controllers (C, C_0) which ensure:

- the *invariance* of a set of states, the *reachability* of a set of states from the initial states of the system, the *attractivity* of a set of states E from a set of states F [47],
- the *minimally restrictive control* (choice of a control such that the system evolves, at the next instant, into a state where the maximum number of uncontrollable events is admissible [47], as well as the *stabilization of a system* [49].

For more details on the way others controllers are synthesized, the reader is referred to [49].

A tool is available, called SIGALI, which implements this with decision diagram [17] techniques typical of model-checking. Some instructions used in the remainder are: `B_True` (resp. `B_False`) which designates the set of states where a predicate is true (resp. false), and `S_Security`, resp. `S_Reachable`, the synthesis operations for objectives of invariance, resp. reachability. The result of synthesis operations is a decision diagram, characterizing the constraints on controllable events necessary for the property to be satisfied (if possible).

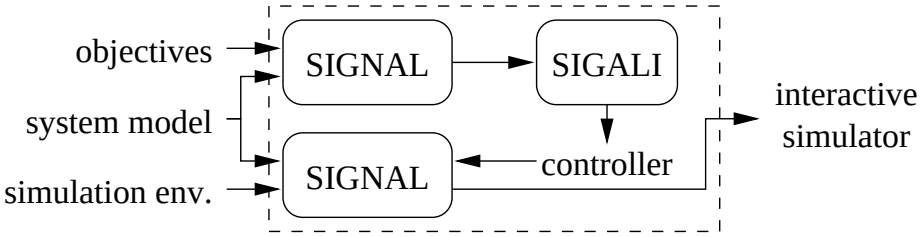

Figure 9: The SIGNAL/SIGALI environment.

Specification, synthesis, and simulation in the SIGNAL/SIGALI environment goes as shown in Fig. 9 [47]. It involves the modeling of the system in all its possible behaviors (desirable as well as undesirable, i.e., those *not* respecting properties) and the specification of properties and objectives (invariance, reachability, attractivity). The properties and objectives can be expressed in SIGNAL (actually SIGNAL+, where call to SIGALI functionalities can be inserted), which eases their specification: they can be stated in terms of the variables and events used in the system model. The SIGNAL compiler is then used to produce a transition system given as input to SIGALI, upon which discrete control synthesis is performed automatically. The resulting controller is produced in a form that is recognized by a generic evaluator, which can be integrated with an application-specific graphical simulation environment. The SIGNAL compiler is used once again for the production of an interactive graphical simulator integrating model and controller.

Modifying the specification and obtaining a new controller can be done automatically, by running the same operations, without having to re-examine the whole controller manually. The result of final synthesis is the maximally permissive controller which, when running in parallel with the system, yields the desired behavior. This means there can remain a level of indetermination w.r.t. the action to take in response to an input, given an internal state. One answer is to take an arbitrary solution in the set of correct controls. A less arbitrary one is to give optimization criteria (e.g., costs of states traversed, or transitions taken ...) to be minimized [49]. Another one is to offer an interactive determination execution scheme, where the choice between all the admissible controls is given to a user [47].

5 A framework for discrete control synthesis in robotics

Our goal is to propose a framework where discrete controller synthesis is applied to robotic and control systems programming. Therefore, this application has to be performed on models representing entities currently manipulated in this domain. It has to be integrated in a programming environment in a user-friendly, systematic, automated way, i.e., typically, which does not require the user to be a specialist in discrete control synthesis methods or model-checking tools. Figure 10 illustrates the position of this new functionality relatively to the other, more classical ones:

- code generation from tasks (at low-level, endless loop), model generation from tasks and mission, properties generation (e.g., in temporal logics from a dedicated, specialized user interface);
- verification of the properties, using e.g. model-checking;
- executable code generation, linked with real-time operating system support.

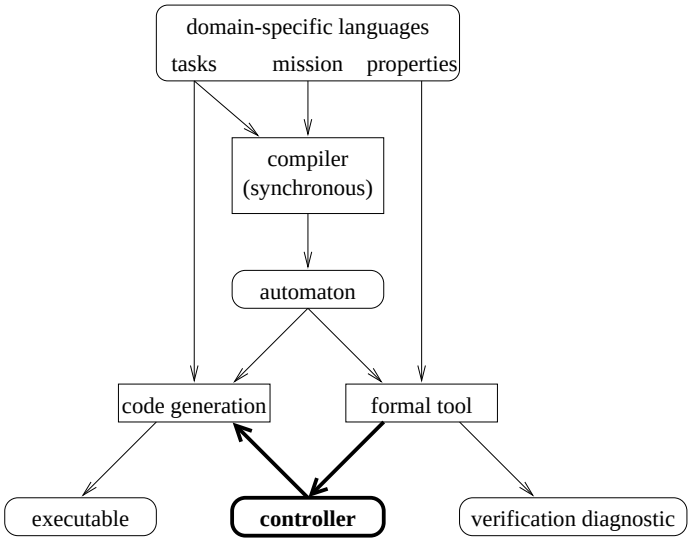

Figure 10: Discrete control synthesis in the programming environment.

In our approach, integrating discrete control synthesis in this framework involves specific questions of:

- specification:
  - modeling the process, i.e. the set of control tasks, possibly partially sequenced into a mission, as predefined structures "instrumented" or "equipped" with the useful controllable events, necessary for appropriate objectives to be controllable;
  - formulating the objectives, possibly defining a specialized language, in terms of the entities manipulated by the user rather than intrinsic to the model, e.g. control events of the control tasks,
- placing the execution of the (possibly interactive) controller within the control architecture (for example in the ORCCAD architecture, placing it among the other tasks and in interaction with them);
- use of the method e.g., how it does constitute a form of teleoperation robotics, in the logical dimension, where from the specification of logical degrees of freedom left to the operator, a controller can be obtained that supervises and disables transitions or trajectories considered dangerous.

In the following, we give a presentation of our preliminary framework exploring these issues, also mentioning possible developments and perspectives. This presentation is made independently of a particular tool support; one possible concrete implementation will be precised further in section 6.

5.1 Specifying behaviors and objectives

5.1.1 Behaviors

Standard tasks. In our discrete model of tasks and missions, we distinguish different discrete control states for each task, as shown in Fig. 11. Initially, a task is `Idle`. It goes from `Idle` to `Act` when there is a request (event `req`) and the controller accepts it (event `go`), i.e. the control constraints allow it. With the intention of "installing" controllability in the model, a `Wait` state has been incorporated to enable the recording of a request when the activity of another task prevents the controller from starting it. The controller may choose to make it active once the conditions are favorable. Termination of a task is signalled by the event `stop`. Under this model, only the event `go` is assumed to be *controllable*; the others are *uncontrollable*.

As we will see further for default tasks, it can also be useful to allow the controller to refuse the termination of a tasks (i.e., it reached its goal, but it is kept active in order to maintain its objective, until another one can be started). Other possible extensions to this model would be representation of the initialization phase

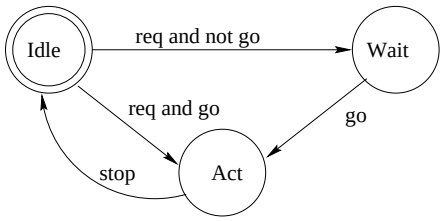

Figure 11: The discrete model of a *standard* task.

or of multiple modes, with different qualities (more or less degraded) and computation costs, the use of several waits and/or of different policies of activation by means of priorities. More elaborate task models could also consider request cancel and preemption structures in missions, and how they would constitute partially controllable stopping of tasks. Also interesting for structured specification of tasks would be the introduction of a notion of hierarchy, a tasks being decomposed into sub-tasks. The use of hierarchy in control synthesis would be a related issue: more structural information can facilitate formal computation.

Default tasks. Robots or control systems often require to be always under control, even for rest configurations, because of gravity or other external forces. This motivates the introduction of a pattern for default tasks, shown in Fig. 12.

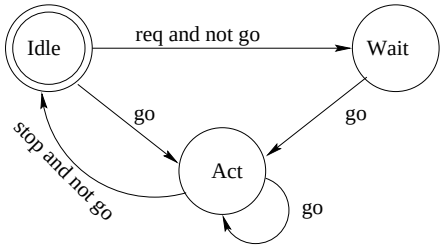

Figure 12: The discrete model of a *default* task.

It is similar to the standard task except that it is not necessary to have a request in order for a default task to become active. However, if a request is made by the environment (e.g., the operator or the user) without the permission of the controller (i.e., `not go`), then the task evolves into the `Wait` state, as before. Also, when the event `stop` is received, the controller decides upon the termination of the task, by triggering `not go`. The meaning of the `stop` event is that it is received at each instant where the task is ready to terminate. For example, in the case of control tasks, it can corresponds to the fact that a control law has reached a point where it is close enough to the goal, so it can go on staying close to it, or terminate in order for the control to proceed with another tassk in sequence. Otherwise, the task remains active. It means that it is the controller which decides whether a task has to be stopped or not. The default task gives the controller more power and restriction possibilities. In the design of a system, safety-critical tasks (e.g. the task of maintaining the current position) are generally modeled as default tasks.

From the particularities of this proposed behavior, it follows that not just any actual task can be associated with the default status. Such a control task must be applicable in all situations, without conditions, and also have an effect on the environment which is compatible with the discrete controller inhibiting its stopping (e.g., maintaining a current situation, such as the position of the robot). It must be interruptible (stoppable conditionally) and startable by the controller, but also as requested by the user or mission.

There are other situations where it can be useful to have a task startable by the controller. An example is that of transitory modes, where two tasks or task modes, implementing two control laws, can not be executed in direct sequence, for reasons due to their physical definition. A transitory mode, possibly specific to the preceding and the following tasks, must be used in between. In this case, the transitory mode is a

tasks that should be started by the controller upon the termination of the preceding one, if the following one is requested. Upon termination of the transitory, the following task is released.

Control missions. Missions are composed of a number of tasks, contributing to achieve the goal of the mission. The sequencing or synchronizations between the tasks are guiding the requests and releases of activity.

In the present state of our framework, missions are obtained by the parallel composition of tasks, which builds the cartesian product of automata. The synchronous composition [27] is defined exactly like that, and therefore using synchronous languages and compilation provides for efficient tool support. This way, we obtain representation of all possible behaviors. Given the looseness of this coupling, all tasks can be activated independently. For n tasks, this automaton is of a size of 3^n.

An interesting extension might be to have a language of "loose" synchronization, where part of the scheduling would be expressed by the user, in the form of a script or a scenario, and synthesis would serve as a completion. To this end, use might be made of concepts like the ones in *GTi* [63], where tasks are defined by associating a data-flow process with a time interval for their activity; differently to their use in direct robot programming [62], they might serve as partial constraint of the parallel composition mentioned above, hence participating in the specification, and reducing the set of behaviors on which controller synthesis is performed. Another possibility is to consider a task-level language like PILOT [66] where a given sequence is defined, resulting is a sequence of requests, themselves controlled, e.g. in an interpreted way, by the automatically synthesized controller, ensuring satisfaction of the appropriate properties.

5.1.2 Objectives

The model, constructed as explained above, includes amongst its configurations some that are undesirable, for example for reasons of resources to be shared (e.g., an actuator between different control laws), or criteria related to the functionality fulfilled by the tasks (e.g., incompatible side effects on the device or its environment). Amongst the paths described by sequences of transitions, there can also be undesirable ones, for example for reasons of necessary transitory modes between some tasks (e.g., between velocity-based and position-based movement control of a motor). In order to control these situations, we have to specify the properties on the states and events to be either achieved or avoided. Then, using them as synthesis objective, we can obtain the discrete controller, if it exists, which will constrain the behaviors in such a way that only those satisfying the properties will be allowed.

Properties and objectives specification. The formulation of objectives involves writing temporal logics formulæ which can be very technical; there are works contributing to making such specifications more accessible, on the basis of domain-specific knowledge and notion, e.g., in robotics, or even finance [31]. There are perspectives for a specialized language for properties as for verification in ORCCAD.

Observers can be used as an alternative. Instead of a predicate on state variables, or a temporal logic formula, they are defined by an automaton, recognizing the sequence, with a terminal state. The global system is the parallel composition of the observer and the pre-existing system. It is submitted to an objective of safety keeping out of the terminal state. This method has the advantage of making it possible to use the same specification language as for behaviors. As an example, to ensure that the system avoids a certain sequence of task activations, one can define a transition system recognizing that sequence, with a final state designating the error. The reachability of this state is then a simple property that can be used for verification or synthesis.

Generic objectives on the tasks. We have identified a few properties and objectives that can be defined on the task patterns proposed above. They are domain-specific in that they concern a particular abstraction of control tasks, and that the constraints are related to requirements of control systems. At the same time, they are generic w.r.t. that domain, meaning that they are relevant to a wide range of applications, and that they can be significantly used for any instantiation of the given patterns. They can also be systematically derived for robotic missions built from task-level components as we introduced. In that sense, we have a

framework where models as well as objectives can be automatically compiled from high-level specifications in a domain-specific language.

They are based on the notion that the system to be controlled is composed of a set of actuators $a \in A$. For each of them, there exists a set of control laws or tasks $t \in T_a$, each defining a different behavior. These tasks are instanciations of the patterns described previously, i.e., for each of them we have the states $idle_t$, $wait_t$, act_t.

Unicity of control for an actuator. A basic property of such a system is that there always is, for each actuator, at least one control law controlling a given actuator (otherwise the actuator is not under control, and may e.g., fall down), and at most one (otherwise the actuator receives possibly contradictory commands).

Invariance objectives can be expressed concerning the state information of the control. We define a Boolean variable in terms of the model state variables (i.e., an observer of the situation), and then define the objective as a simple expression, like: achieving invariance of the truth of the value of that Boolean.

Existence at all instants of a control law can be described as: for the set of actuators A, and the set of tasks T_a potentially controlling the actuator $a \in A$:

$$\bigwedge_{a \in A} \left(\bigvee_{t \in T_a} act_t \right)$$

The objective is to *make it invariantly true.*

Exclusivity of control laws on the same actuator is another basic property to be maintained in a control system, i.e., at most one control law at a time can be controlling a given actuator. It can be achieved in a way quite close to the previous one: it is a property on states, where the situation *to be avoided* is that for any actuator there is more than one active task, in other words:

$$\bigvee_{a \in A} \left(\bigvee_{t,t' \in T_a, t \neq t'} (act_t \wedge act_{t'}) \right)$$

The objective is to *make it invariantly false.*

Return to an initial state. Another interesting global property to check or insure is that the system can always be returned into an initial configuration. Reachability can be used to specify this, by characterizing the states where that configuration holds, and *making always reachable* the set of states where that predicate holds.

Tasks interactions and incompatibilities. Other properties concern the interactions between different tasks or subsystems. Safety concerns can command, e.g., to exclude manually controlled movements of an actuator when others in the same volume are in movement. These properties are less generic than the previous ones, in the sense that they are architecture-specific. However, they describe patterns that can be re-used in a wide range of applications.

Such a pattern can concern incompatibility of activity modes of different tasks, possibly concerning different actuators, but which should not be active in parallel at the same time, due to resource sharing, which can be taken in a general sense as e.g., external, physical reasons related to their effect on the environment. The corresponding synthesis objective is then to *make invariantly false* the predicate making the conjunction of concerned activities, e.g.,

$$act_t \wedge act_{t'}$$

The incompatibility can be conditioned by external features, in which case, the property, involving that conjunction, features an implication as in e.g., $cond \Rightarrow \neg(act_t \wedge act_{t'})$, i.e., $\neg cond \vee \neg(act_t \wedge act_{t'})$ which must be made invariantly true, or alternately, the "wrong" situation characterized by

$$cond \wedge (act_t \wedge act_{t'})$$

which must be *made invariantly false.*

Task sequences and transitory modes. Until now properties considered are static in the sense that they concern situations rather than series of transitions. More dynamical properties are related to allowed task sequences and requested transitory modes. As was said before, this can involve defining observers recognizing these sequences, and distinguishing the terminal state (or set of them), in order for it to be used in objectives of reachability or invariance.

Inhibiting two successive activations of the same task t_1 without carrying another one t_2 in between is an example of property. More precisely, we want to have, between the ending of t_1 and its next activation, at least a complete activation of t_2, from activation to end. Cases with simultaneity are acceptable. For this, an observer can be proposed as in Figure 13.

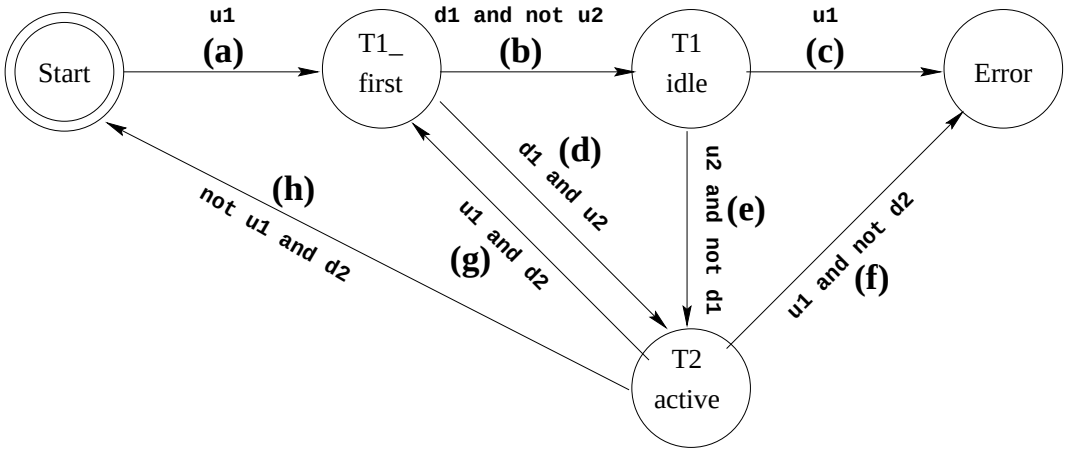

Figure 13: An observer process for two successive activations of the same task.

This automaton, initially in state `Start`, observes changes in the value of state variables act_{t_1} and act_{t_2}, i.e., we have u_1 when act_{t_1} goes up from false to true (i.e., t_1 becomes active), and d_1 when act_{t_1} goes down from true to false (i.e., t_1 becomes inactive). Along the same lines, we have u_2 and d_2 for act_{t_2}.

So we have the following transitions:

(a) a first occurrence of the starting of task t_1 causes a transition to state `T1_first`.

Upon the task deactivation d_1:

- (b) in the absence of the activation of act_{t_2}, we go to state `T1_idle`.

  There,

  - (c) if task t_1 goes active again before an activation of t_2, the observer goes into the state denoting the error: `Error`.
  - (e) if act_{t_2} becomes active: u_2, and t_1 does not go active again, then we go to state `T2_active`.
- (d) if simultaneously act_{t_2} becomes active: u_2, then we go to state `T2_active`.

  There,

  - (f) if t_1 goes active again before the end of t_2, the observer goes into the state denoting the error: `Error`.
  - (g) if t_1 goes active again simultaneously with t_2 ending, then we go to `T1_first`.
  - (h) if t_1 does not go active again when t_2 ends, and then the observation starts from the initial state again.

The desired property is that a sequence reaching `Error` can not be followed. The way to achieve that is to synthesize a controller for the objective: making invariant the value false for the state variable `Error`.

Avoiding of a direct sequence of one task into another without a third one in between is another example. This corresponds to the existence of control laws which needs to be separated by a special mode handling the transitory situation between the two (e.g., velocity control followed by position control). We want this transitory task or mode t_3 to be started automatically (i.e., without request, as in the case of default tasks) by the synthesized controller, when the first task t_1 stops in the presence of a request for the following one t_2. The task t_2 would then be released and activated only upon the stopping of t_3.

Let us consider the particular case of an actuator for which we have just these three tasks, t_1 and t_2 standard ones, and t_3 a default task. The default task is the one that can be started by the controller without external request. We already have specified that at least and at most one ¡etc ...¿, hence the observer concerns only cases satisfying these properties, and we just have to ¡etc ...¿.

We already have specified that at least and at most one task must be active at every moment, hence the observer concerns only cases satisfying these properties, and we just have to specify that we do not want t_1 and t_2 to be in direct sequence. Then, t_3 will be started according to the previous objective. An observer characterizing these situations is proposed in Figure 14.

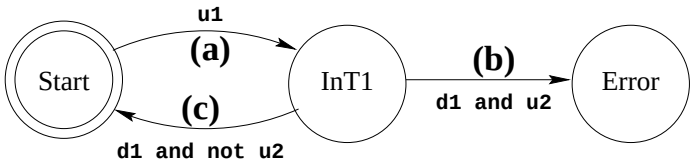

Figure 14: An observer process for transitory tasks.

This automaton, initially in state `Start`, observes (a) an occurrence of the starting of task t_1 (when act_{t_1} becomes true) and goes to state `InT1`. There, either (b) task t_1 goes inactive again while the other task t_2 goes active: then the observer goes into the state denoting the error, or (c) task t_1 goes inactive but the other one t_2 is not activated immediately, in which case the observation starts from the initial state again. The desired property is that a sequence reaching `Error` can not be followed. As above, the way to achieve that is to synthesize a controller for the objective: making invariant the value false for the state variable Error. A comment related to this example is that having a class of tasks startable by the controller is meaningful here too, but in a way different from the case of default, initial tasks. Another one is that the possibility of having requests of other tasks during t_3 raises the question of managing priorities between waiting tasks, i.e., here, to know whether t_2 is released in priority or not.

Related to this last point, an interesting extension would be to characterize objectives about waiting tasks and defining policies for their order of release, which would be implemented by the synthesized controller.

Architecture-specific and mission-specific objectives. The properties considered previously, and the objectives with them, were not specific to anything else than the notion of task which was proposed. Therefore, their design and formulation does not depend on particular architectures (in terms of sensors, actuators, and set of tasks), or on particular missions defined for being executed on such an architecture. Their use for the completion of a partial mission specification into a complete controller is a step towards a form of mission programming organized around tasks rather than around physical (or other) resource management. This latter aspect, related to constraints between resources, is handled automatically, providing the designer with automated and correct results, and releasing concentration from technicalities not central to the application area.

However, more knowledge about the internals of tasks, the nature of sensors and actuators, can of course lead to specific properties and objectives. In quite the same way, some aspects to be verified and controlled pertain to the specific mission considered, i.e., they are related to the special sequencing of tasks constituting the mission. At the architectural level, it seems interesting to determine properties that are meaningful independently of the mission. They have to be formulated once, and can then be reused without having to re-define them formally. At the mission level, however, properties can be but specific, and must be formulated formally by the mission designer, i.e., the end-user. Therefore there is a need for a domain-specific language for the expression of properties, typically making end-users feel less uneasy than with temporal

logics. Attempts exist in that direction, related to the MAESTRO language [24] in robotics, or to the domain of management [31]. One of the problems is to find a language level combining specialization, easing the use by application domain specialists, and expressivity, making a good use of the available model.

5.2 Obtaining the controller

Having built a model of the uncontrolled system, and determined objectives to be achieved by the controller, the next step is to automatically produce a controller. As was said in Section 3, the theoretical background is mature, algorithms have been designed, and effective tools exist supporting this kind of computation. In our framework, and seen from the point of view of a user, it amounts concretely to encoding the model in a format recognized by a synthesis tool, and invoking the appropriate functions of that tool.

As a preliminary testbed on the way to an environment as pictured previously in Figure 10, we have built up an experimental set-up where we follow these phases from specification to interactive execution. It is based on SIGALI and SIGNAL, and described further in Section 6.

5.3 Using the controller

The controller is synthesized in the form of a relation defining, with respect to a given state and input (a set of uncontrollable values), a set of controllable events values that satisfies the objective. In the particular case where it is a function, it uniquely defines, from the inputs, the behavior of the controlled system.

5.3.1 Simulation

The controller can be used in simulation, as a tool for exploring the controlled behavior, e.g., in relation with a model of the system also encompassing non-discrete aspects. From this exploration, Boolean abstractions of more properties can be defined, in order to bring a larger part of the behavior into the discretely controllable realm.

5.3.2 Execution

The obtained controller can also be embedded into an execution architecture, as a task control component. It can be made a function, by e.g.,

- having a random choice of a correct control among the set proposed by the relation, or
- having a choice using some local optimization criterion, evaluated at run-time.

5.3.3 Interactive execution as a form of safe discrete teleoperation

One aspect interesting us particularly, with respect to robotics as our application domain, is interactive execution as a form of safe discrete teleoperation. The issue to be considered is that interactive systems, and in particular teleoperated systems in their discrete aspects, are increasingly complex, which brings difficulty and error-proneness not only in their design, but also in their operation. The problem is to be at once interactive *and* safe. The techniques presented above have a potential to constitute a form of discrete form of teleoperation robotics, in the logical dimension, where from the specification of logical degrees of freedom left to the operator, a controller can be obtained that supervises and disables transitions or trajectories considered dangerous.

Indeed, teleoperation or telerobotics is characterized by a (variable) degree of manual interaction in the control of the system. This is motivated by e.g., operations for which no automated control is currently available, activities intrinsically un-automatable, error and accident treatment modes where the system has to be brought back into a state from which automation can take over, modification or update of controllers by maintenance staff, different from the design or specification staff (as can be witnessed in manufacturing). A typical example is that of a teleoperated articulated arm, carrying a drill; the operator is manually commanding the movement of the arm. When approaching the location of the hole to be drilled, movement can be completely determined manually. When in position to drill, however, it is a useful assistance to go

into a mode where only movement along the axis of the drill is manually controlled, the other, lateral ones being controlled automatically. This share of the control between automation and operator decision can be transposed in our framework.

If here we consider telerobotics on the discrete side, concerning events typically starting or stopping tasks, then the interface mentioned above constitutes the kernel of a controller coming as a complementary to the user's commands, and keeping them within a safe state space. Although this vision of teleoperation is not usual, we feel that it corresponds to a reality, and constitutes an answer to the seemingly conflicting requirements of safety and interactivity.

This mechanisms could be interestingly coupled with high-level languages for teleoperation [54, 60, 23, 66]. There, a language is used in order to define a sequencing of actions, with possibilities for run-time modification, which however make it impossible to verify correctness off-line. Making such a language the producer of task activation requests, submitted to the controller, would bring a layer improving the safety of the global system.

This basic interactive execution scheme has to be instrumented (e.g., with alarms and warnings to the operator) and put into a run-time executive framework. This integration can then raise other questions related to other aspects of teleoperation (the continuous part, the communications, ...).

5.4 Integration in a programming environment

Our framework is generic enough to design and develop an automated process on these bases, along the lines of Figure 10, and more precisely illustrated in Figure 15.

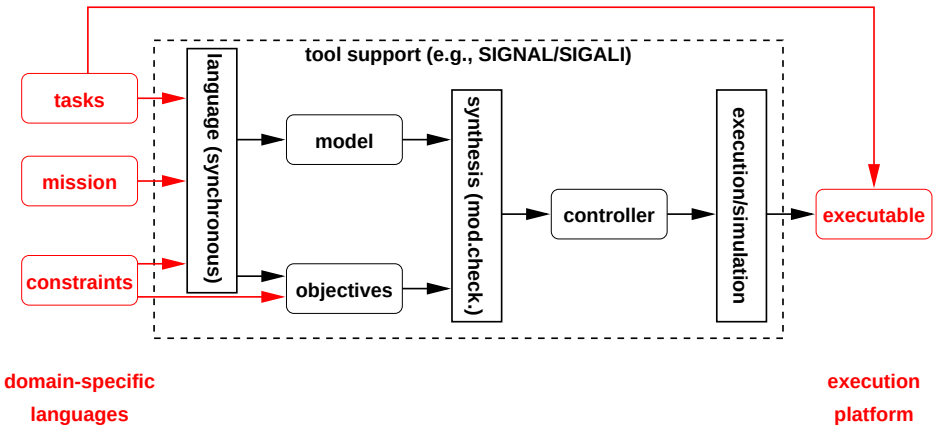

Figure 15: Integration of discrete control synthesis into an automated process.

Practically, it will consist of

- the compilation (relying on, e.g., synchronous techniques) of mission and task models, instrumented for synthesis like in Figures 11 and 12, into a global automaton,
- the construction of a set of synthesis objectives, from pre-defined properties related to the task structures,
- a call to the formal tool performing synthesis, to produce the controller.
- integration of the synthesized controller into an execution framework.

As a preliminary testbed on the way to such an environment, we have built up an experimental set-up described in Section 6, where we follow these phases from specification to interactive execution.

6 Implementing the framework with Sigali and SIGNAL

We use the design environment based on the synchronous language SIGNAL. The specification of tasks is done in SIGNAL. We use its associated formal computation tool SIGALI, based on symbolic model-checking techniques [48]. The motivation for this choice is directly related to our goal in synthesizing discrete controllers within a programming and execution framework. SIGALI and SIGNAL provide, to our knowledge, for the most complete methodology and tool-support environment available [47], including the execution of the synthesized controller. As a synchronous language, it is in principle compatible with the others. In particular, the relation with ESTEREL, hence with ORCCAD, is that ESTEREL can be compiled into a format called DC, which can in turn be accepted by the SIGNAL compiler, and then the SIGALI tool [13].

6.1 Specification

6.1.1 Tasks

This section gives the SIGNAL code for the task patterns defined in Section 5.

State memorization. A first process is defined in order to have the appropriate Boolean state memorization element. The process `m` has a parameter giving the initial value `Minit`. It has two Boolean inputs `Mtrue` and `Mfalse`, indicating respectively, when they are true, that the memory is assigned to `true` or `false`. This is encoded in the equation defining `NewM`. When a new value is present, then it is passed through `M`, else the previous value `ZM` is used.

```
process m = { boolean Minit; }
     ( ? boolean Mtrue, Mfalse;
       ! boolean M, ZM;)
   (| (| NewM := (when Mtrue) default (not (when Mfalse))
       | M := NewM default ZM
       | ZM := M$1 init Minit |)/ NewM |)
   where
      boolean NewM;
   end;
```

Table 3: SIGNAL code for state memorization.

It can be noted that this encoding of a register is not the simplest one imaginable, but that the way everything is made explicit here brings facilitation of some technical aspects in the simulation interface.

Control tasks. A control task is modeled as a three-state automaton as in Figure 11. This can be encoded

```
process tc = { boolean idle_init; }
     ( ? boolean go, req, stop;
       ! boolean idle, act, wait;)
   (| (| (idle,z_idle) := m{idle_init}(make_idle,req)
       | (act,z_act) := m{not idle_init}(start,stop)
       | (| wait := not (act or idle)
          | z_wait := not (z_act or z_idle) |)
       | start := ((when req) when go when z_idle) default (go when z_wait)
       | make_idle := stop when z_act
       | idle ^= req ^= stop ^= go
       |)/ make_idle, start, z_wait, z_act, z_idle |)
   where
   process m = ...
   boolean z_idle, z_act, z_wait, start, make_idle;
   end;
```

Table 4: SIGNAL code for the control task.

in SIGNAL as in Table 4. Input signals are Booleans go, req and stop. Outputs are Boolean indicating for each of the three states whether it is true or not. A Boolean parameter idle_init is used to initialize the state as being, when true: idle, else act. The three states are encoded using two registers as defined above in Table 3. One state variable is idle, becoming false when req is true, and true when make_idle is, i.e., when stop is true when the former state was act. The other one is act becoming false when stop is true, and true when start is, i.e., when req and go are from state idle, or when just go is from state wait. The third state, wait, is defined in terms of the two previous variables, i.e. it is true when none of the others is; its previous value z_wait is defined the same way.

Default tasks. Their encoding in SIGNAL, shown in Table 5, follows quite the same pattern as that for control tasks, except for:

- a conditional deactivation and make_idle (go must be false),
- a simplified start (no req necessary).

```
process tc_default = { boolean idle_init; }
     ( ? boolean go. req, stop;
       ! boolean idle, act, wait;)
   (| (| (idle,z_idle) := m{idle_init}(make_idle,req or go)
       | (act,z_act) := m{not idle_init}(start,stop when (not go))
       | (| wait := not (act or idle)
          | z_wait := not (z_act or z_idle) |)
       | start := ((when go) when z_idle) default (go when z_wait)
       | make_idle := stop when z_act when (not go)
       | idle ^= req ^= stop ^= go
       |)/ make_idle, start, z_wait, z_act, z_idle |)
   where
   process m = ...
   boolean z_idle, z_act, z_wait, start, make_idle;
   end;
```

Table 5: SIGNAL code for the default task.

Control missions. When having constructed a library of such tasks for a given systems, by instantiating the previously defined patterns, a control mission can be specified simply by using the SIGNAL composition of processes.

For example, one way of structuring the specification is to have, for each actuator, a composition of tasks representing each available control for that actuator. This constitutes a sub-system; such sub-systems can assembled into larger systems, hierarchically.

6.1.2 Properties and objectives

Properties are defined using Boolean expressions and signals, as well as observer processes which can be constructed using the same memory process m as defined in Table 3.

Generic objectives on the tasks. The various generic properties proposed in Section 5.1.2 are encoded in SIGNAL and SIGALI quite directly, as follows.

Unicity of control for an actuator. Table 6 shows the Booleans defined for an actuator called arm, with tasks maint_arm, manu_arm, home_arm, and auto_arm. The first one, Ctrl_arm_act, is true when at least one task is active. The second one, Ctrl_arm_red, is true when more than one task is active. The third one, Ctrl_arm, is true when a single task is active.

The synthesis objective to be used by SIGALI is specified in the last line: it consists of making invariant (S_Security) the trueness of Boolean Ctrl_arm.

```
| Ctrl_arm_act := Act_maint_arm or Act_manu_arm or Act_home_arm or Act_auto_arm
| Ctrl_arm_red := (Act_maint_arm and Act_manu_arm) or (Act_maint_arm and Act_home_arm)
                or (Act_maint_arm and Act_auto_arm) or (Act_manu_arm and Act_home_arm)
                or (Act_manu_arm and Act_auto_arm) or (Act_home_arm and Act_auto_arm)
| Ctrl_arm := Ctrl_arm_act and (not Ctrl_arm_red)
| SIGALI(S_Security(B_True(Ctrl_arm)))
```

Table 6: SIGNAL code for the unicity of control for an actuator.

This code is the one for the model of the sub-system `arm`; a complete system can be constructed by instances of similar patterns for the different actuators of the system. It can be noted that here, the properties and objectives local to a subsystem can be specified locally to the corresponding SIGNAL process. This facilitates modularity and reusability of specifications.

Return to an initial state. Table 7 shows the code scheme defining a Boolean `init_state`, characterizing the corresponding state, and the way the objective is encoded for SIGALI, of making states where this Boolean is `true` always reachable form the current state.

```
| init_state := ...
| SIGALI(S_Reachable(B_True(init_state)))
```

Table 7: SIGNAL code for the return to an initial state.

Tasks interactions and incompatibilities. For making exclusive the activity of two incompatible tasks `t1`and `t2`, a Boolean can be defined as in Table 8.

```
| incomp := Act_t1 and Act_t2
| SIGALI(S_Security(B_False(incomp)))
```

Table 8: SIGNAL code for Tasks interactions and incompatibilities.

Task sequences and transitory modes. These properties, and more generally the ones involving an observer defining a state `Error` which we want not to be reached, are treated by writing an observer process, using `m`, and with an output Boolean `Error` which is `true` when the corresponding state is entered. Table 9 shows how this Boolean is used in the objective for SIGALI.

```
| obs{}
| SIGALI(S_Security(B_False(Error)))
```

Table 9: SIGNAL code for an observer with an error output.

For the example properties described in Section 5.1.2 in Figures 13 and 14, the coding of corresponding observers is detailed in see Section 7.1.2 for particular instanciations.

6.2 Synthesis

To perform the computation of the controller with regard to the different control objectives, the SIGNAL compiler produces a file which contains the PDS resulting from the abstraction of the complete SIGNAL program and the algebraic control (as well as verification) objectives. We thus obtain a file that can be read by SIGALI. Suppose that we must enforce, in a SIGNAL process named "system" the invariance of the set of states where the boolean `PROP` is *true*. The corresponding SIGNAL program is :

```
(| system{}                                  system modelled in Signal
 | PROP := ...                              definition of the boolean PROP in Signal
 | Sigali(S_Invariance(B_True(PROP))
 |)
```

The corresponding SIGALI file, obtained after the compilation of the global SIGNAL program, is the following:

```
read(''system.z3z'');                       loading of the PDS "S"
Set_States: B_True(PROP);                   states where PROP is true
S_c: S_Invariance(S,Set_States);            synthesize the controller ensuring the invariance of Set_States
```

The file "system.z3z" is the PDS that represents the initial system. The PROP signal becomes a polynomial Set_States expressed by state variables and events, which is equal to 0 when PROP is *true*. The last line of the file consists in synthesizing a controller which ensures the invariance of the set of states where the polynomial Set_States takes the value 0. This file is then interpreted by SIGALI that checks the verification objective and computes the controller. The result of the controller synthesis is a polynomial which is represented by a BDD (Binary Decision Diagram). This also produces one file with the controller in an internal representation, and another file with information for the integration into a simulator. For these two files to be produced, the specification must feature the code line:

```
| SIGALI(Simul())
```

6.3 Interactive simulation

Further, to obtain a simulation that allows to visualize the new behavior of the controlled system, the controller is automatically integrated[7] in the initial SIGNAL program through an algebraic equation resolver written both in SIGNAL and C^{++}. At the same time, the user has the option of adding in this new program some generic processes of simulation. These SIGNAL processes perform, after compilation, the automatic construction of graphical input acquisition buttons and output display windows for the signals of the interface of the program, in e.g. an oscilloscope-like fashion; with regard to the commands and the uncontrollable events, the graphical acquisition button processes are automatically added in the SIGNAL program when the resolver(s) is (are) included. From this point, it is sufficient for the user to compile the resulting SIGNAL program which generates executable code ready for simulation. Once the controller has been computed and integrated in the new SIGNAL program as explained in the previous section, we have to simulate the result of the synthesis.

In most cases, the controller is not deterministic, in the sense that for a given state and a given uncontrollable event, more than one command can be admissible. To solve this problem, we choose to perform a step by step simulation in an interactive manner. During the first stage of SIGNAL program specification, the user indicated the controllable and uncontrollable inputs. Thus, the values of the controllable events will be chosen by the user under the control of the resolver through an interactive dialogue box. For example, when the user makes a choice (i.e., the user chooses a particular value for one of the commands), this choice is automatically sent to the resolver, which returns the set of possible values for the remaining commands. In fact, each time a new choice is made by the user, a new controller is computed, in the sense that one variable of the polynomial controller has been instantiated. New constraints can then appear on the commands which are not totally specified (there still exist more than one choice); During this exchange between the dialogue box and the resolver, some commands can be totally specified by the resolver in which case their values are then imposed.

The choice of the command values can be performed step by step by the user, or using a random process for a step of simulation. In the second case, the resolver chooses the command values. The user can also ask for a random simulation during an indeterminate number of simulation steps. A graphical example of such a simulation is given in Section 7.4.

[7] based on the two files generated using the SIGALI command `Simul()`.

7 Application to an excavation system

We consider a simplified model of an excavator system, inspired by the application domain of the IST project TELEDIMOS [65], hoping for simultaneous illustrativity, simplicity and brevity. It provides us with an example which is manageable and overseeable hence illustrative, and at the same time showing some complexity [55].

7.1 The excavation system

The system we consider decomposes into sub-systems, according to the actuators:

- an articulated arm,
- a grip held at the end of the arm (which can be used for transferring material from one place to another or for digging),
- a rotating cabin on which the arm is mounted,
- and a mobile base , carrying the whole, itself composed of two tracks.

Each subsystem i.e., actuator, is equipped with a library of control tasks, corresponding to different functionalities of the device, and different ways of achieving them, according to different criteria. The complete excavator system is simply constituted by the composition of all its actuators, each with its control tasks, as shown in Fig. 16. It can be controlled completely manually, using handles, pedals and/or a steering wheel. The operator traditionally has to take care for the feasibility of the tasks combinations he controls; e.g., nothing prevents him or her to press simultaneously several handles or pedals. For example, it is not impossible to press at the same time acceleration and brake. Even more intricate is the care to be taken of the sequencing of controls, where nothing prevents the operator from going from a velocity control to a position control on the same engine, or another such problematic transitory mode. It can also be controlled in a more or less automated way, where control laws are in charge of the movements. In these cases, some combined movements considered dangerous when performed manually (e.g., driving while manipulating the articulated arm) can be allowed, provided the control laws are valid.

7.1.1 The different actuators

As we said above, the excavation system is composed of several sub-systems. For each of them there is a set of tasks that can control it: they take input from some defined sensors and output commands towards the subsystem seen as an actuator. Each of the elements of the excavator has to be assigned a default control task. The purpose is to avoid situations where a subsystem in not under the control of any task. In the following, the default task will consist in maintaining the current position at the moment when the task is started. There can also be tasks considering subsets (possibly the whole system) as an actuator. The Excavator System with its actuators and the atomic tasks within each actuator are shown in Figure 16.

The Grip. The grip is equipped with two tasks:

- grip manipulation (called `grip-manipulate`). It is a standard task. This task is supposed to be manual i.e., the environment (the operator or the user) manipulates the grip. At the lower level, the goal of this task is to either open or close the grip.
- maintain a position (`grip-maint`). It is a default task. It maintains the grip at the current position.

Articulated arm. We consider four tasks for the articulated arm:

- manual control (called `arm-manu`), where movements of the arms are determined by the operator directly, manually manipulating handles (possibly with force-feedback, or sharing of degrees of freedom with an automated control law in computer assisted teleoperation);

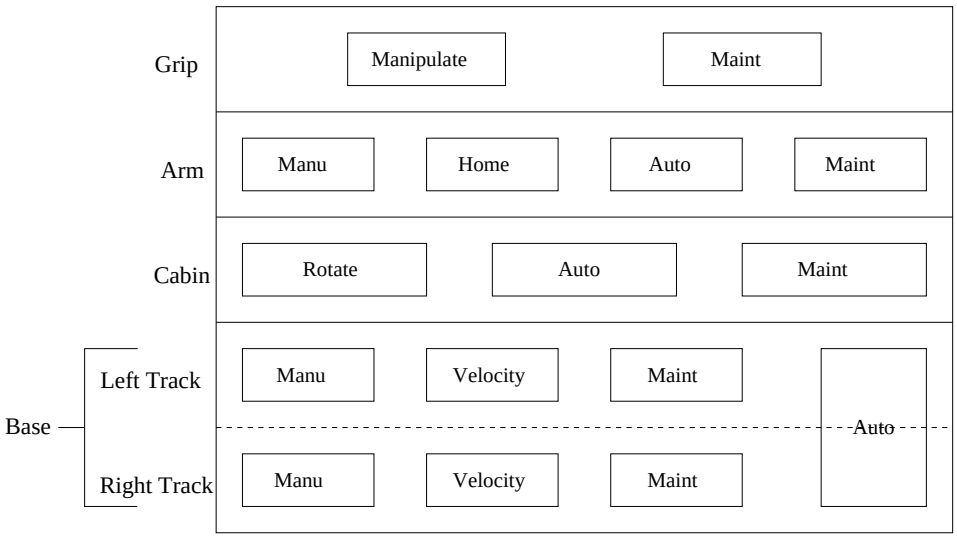

Figure 16: The model of the Excavator System.

- go to home position (`arm-home`), where the arm is moved from any current position to a predefined position (e.g., where it can be mechanically locked, or where it can rest with less energy consumption);
- automated move (`arm-auto`), where movements of the arms are determined by an automated controller, implementing some control law: there can actually be a library of such tasks, with e.g., position control, trajectory following, speed control, use of different sensors (telemetry, vision, ...);
- maintain a position (`arm-maint`), at the current (articular) position of the arm at the moment when the task is started; it can be noted that this constitutes an actual control not accounted for by mere power cut-off: in particular this task compensates for gravity pulling the arm down, or strong wind or water current.

Rotating cabin. The Rotating Cabin is composed of three different tasks.

- rotating movement (called `cabin-rotate`). This is the operator who manipulates the movement of the cabin by means of handles or buttons. The rotating aspect (i.e. a clockwise movement or a counter-clockwise movement) is somehow "included" in this task and depends on the handles or buttons manipulated by the operator. However, at our level of abstraction this has to be considered a single task, as it only involves one control law.
- automated move (`cabin-auto`) where movements of the cabin are determined by an automated controller, implementing some control law: here too there can be a library of such tasks;
- maintain a position (`cabin-maint`) at the current (angular) position of the cabin at the moment when the task is started; in particular this task can compensate for a rotation induced, when on the slope of a hill, by gravity pulling the arm down.

Mobile base. The tasks for the mobile base are defined symmetrically for each of the two tracks, right and left:

- velocity regulation (called `base-l-velocity` and `base-r-velocity`) that corresponds to the "speed" of the base. It can be different for each track, involving a turning movement of the base. It is determined by the operator directly, using handles and/or pedals;
- manual control (`base-l-manu` and `base-r-manu`). Same as the arm manual task.

- maintain position (`base-r-maint` and `base-l-maint`) at the current (odometric) position of the base at the moment when the task is started; in particular this task can compensate for gravity when on the slope of a hill.

There also exists an automated movement task (called `base-auto`) which is common for both the tracks. If this task is active, then the controller takes control of both the tracks in order to achieve a particular goal (again there can exist a library of such goals).

Missions. In the present state of our framework, missions are obtained by the parallel composition of tasks, which builds a Cartesian product of automata. Thus, the representation of all possible behaviors is obtained.

The model includes amongst its configurations some that are undesirable, for example for reasons of resources to be shared (e.g., an actuator between different control laws), or criteria related to the functionality fulfilled by the tasks (e.g., incompatible side effects on the device or its environment). Amongst the paths described by sequences of transitions, there can also be undesirable ones, for example for reasons of necessary transitory modes between some tasks (e.g., between velocity-based and position-based movement control of a motor). In order to control these situations, we have to specify the properties on the states and events to be either achieved or avoided. Then, using them as synthesis objective, we can obtain the discrete controller, if it exists, which will constrain the behaviors in such a way that only those satisfying the properties will be allowed.

7.1.2 Properties

Independently of the environment within which it is used, each subsystem can have constraints. The most basic ones concern presence and uniqueness of control. Moreover, if the automated mission controlling the excavator does not specify all the controls of all the actuators at all times, it can be useful to specify that in case no other control is active, then the default control law should be activated. Hence, for each actuator,

1. *It must always be under control (otherwise unpredicted movements can occur).*
2. *There must be at most one active control law (otherwise control can incoherent).*

More interestingly, the architecture grouping each of these subsystems in an excavator bring possible interactions between different subsystems upon which some properties have to hold. manually controlled movements of an actuator when the others upon which it is mounted are in movement. This decomposes into:

1. *No cabin movement (except automated) while driving (i.e. when the base is moving).*
2. *No arm movement (except automated) while driving or rotating.*
3. *No manual manipulation of the grip when arm movement, driving or rotating.*

All properties mentioned above are what is called static properties, in the sense that they concern properties on states, not on successions of events. Interesting properties to ensure would be the following:

1. *exclude two successive manipulations of the grip without any cabin or base movement in between.*
2. *between any two task where the arm moves, the default task should be activated.*

7.2 Model specification in SIGNAL

The tasks of each actuator are specified as SIGNAL processes encoding the behavior of Figure 11 (if it is a control task) as described by Table 4 or Figure 12 (if it is a default task) as described by Table 5, with appropriate input signals. Each of them also has outputs giving the internal state, amongst which the Boolean `Act_`t telling whether the task t is active. As an example, Figure 17 shows the specification of the grip in the SIGNAL graphical editor, with one instanciation of the `tc` process for `manipulate`, and one of the `tc_default` process for `maint`.

Finally, The mission specification is simply the parallel composition of the actuator processes, which is written as in Table 10, where it is also combined with the observer process defined further.

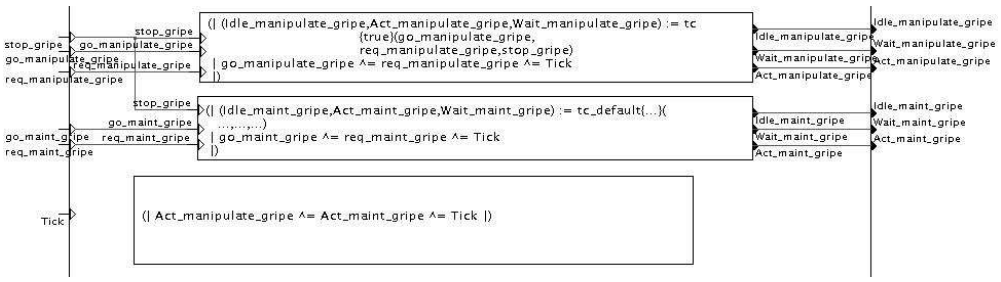

Figure 17: Specification of the grip in the graphical editor.

```
(| obs{}
 | base{}
 | arm{}
 | cabin{}
 | gripe{}
 |)
```

Table 10: SIGNAL code for the mission, composing the tasks for actuators, as well as an observer.

7.3 Controller synthesis with SIGNAL and Sigali

The properties here are quite the same as those encountered in the activity of validating missions using verification. However, their use here, in a controller synthesis setting, makes them into a partial specification for obtaining the correct controller. Let us go through the various objectives the excavation machine system has to verify:

7.3.1 Transversal objectives and properties

The *existence at all instants* of an active control law for each actuator has been described in Table 6 for the case of the arm actuator. Let `Ctrl_arm` be the boolean which is true whenever the property is verified.

Analysis Both the model of the system and the objective are compiled using the SIGNAL compiler, which can generate a file, encoding the model of the mission as a transition system. This file is accepted by SIGALI, and can be submitted to examination using predefined verification functionalities. For example, we can check whether it is possible to reach, from the initial state, the states where `Ctrl_arm` is true, which is done using the SIGALI command:

```
Sigali : Reachable(B_True(Ctrl_arm));
```

which appears to be true, hence it is possible; the next question is whether it is invariant, i.e., always true:

```
Sigali : Always(B_True(Ctrl_arm));
```

which appears to be false, hence some control should be exerted to make it true. Finally, we want to know whether it is possible to act on transitions in such a way as to remain in states where the property holds:

```
Sigali : c_invariant(S,B_True(Ctrl_arm));
```

which appears to be true. This verification phase can be seen as a way to analyze the specification for its controllability. However, the last property tells us that if all events were controllable, there would exists a control satisfying the objective.

Controller Synthesis The computation of the controller itself has to consider that only part of the events are controllable, i.e. `req` and `go` for each task. It is done using the command:

```
Sigali : S_c: S_Security(B_True(Ctrl_arm));
```

which produces a new transition system `S_c`. One more verification on this new transition system confirms that the controller could be produced:

```
Sigali : Always(B_True(Ctrl_arm));
```

Another equivalent, more incremental approach would have been to first synthesize a controlled system S_1 verifying the first objective (i.e. ensuring the *trueness* of the predicate `Ctrl_arm_act`), and then restricting it further so as to verify the second objective (i.e., the predicate `Ctrl_arm_red`). Same controls have to be applied on each actuator of the system.

7.3.2 Objectives Across Subsystems.

We first here consider static control objectives only involving (des)activation of actuators according to the current active tasks of the other actuators. Among others, we computed controllers satisfying the following properties:

1. No cabin movement (except automated) when the base is moving:

```
| Driving := not (Act_maint_track_left and Act_maint_track_right)
| P1 := Act_rotate_cabin and Driving
| SIGALI(S_Security(B_False(P1)))
```

2. No arm movement (except automated) when the base or the cabin is moving:

```
| Rotating := not Act_maint_cabin
| P2 := Act_manu_arm and (Driving or Rotating)
| SIGALI(S_Security(B_False(P2)))
```

3. No manual manipulation of the grip when the base is moving:

```
| Arm_move := not Act_maint_arm
| P3 := Act_manipulate_gripe and (Arm_move or Driving or Rotating)
| SIGALI(S_Security(B_False(P3)))
```

The initial state of the system is that in which all the default position maint tasks of the subsystems are active and all other control tasks are idle. An interesting control objective is to ensure the reachability of a state where all and only the maint tasks are active. Note that this state is not exactly the initial state since some of the other tasks may be in the `Wait` state. But it can characterize a "*restart*" state. We try to ensure the reachability of this "*restart*" state and not the actual one in order not to make the controlled system too restrictive. Specifying the reachability of this state is as follows:

```
| restart := Act_maint_grip and Act_maint_arm and Act_maint_cabin and
        Act_maint_track_left and Act_maint_track_right
| SIGALI(S_Reachable(B_True(restart)))
```

We want to exclude two successive manipulations of the grip without any cabin or base movement in between. To do so, we make use of an *observer* as described in Table 11. It encodes in SIGNAL the instantiation of the observer of Figure 13, where t_1 is `manipulate_gripe`, and t_2 is any non immobile task, i.e., given that always at least one task is active, it corresponds to cases where the cabin and the mobile base (each of its two tracks) are *not* all controlled by their respective `maint` tasks.

Another property, for which the observer is described in Table 12, is that *between any two task where the arm moves, the default task should be activated.* It encodes in SIGNAL the instantiation of the observer of Figure 14 where t_1 is `auto_arm`, and t_2 is `home_arm`; the default task t_3 which will be activated as a transitory is `maint_arm`.

Each resulting `obs` process has a Boolean output `Error`. The following SIGNAL+ code causes computation the corresponding controller.

```
| obs{}
| SIGALI(S_Security(B_False(Error)))
```

```
process obs1 =
     ( ? boolean Act_manipulate_gripe;
         boolean Act_maint_cabin;
         boolean Act_maint_track_left;
         boolean Act_maint_track_right;
       ! boolean Error;
     )
   (| ZAct_manipulate_gripe := Act_manipulate_gripe$1 init false
    | u1 := Act_manipulate_gripe and not ZAct_manipulate_gripe
    | d1 := not Act_manipulate_gripe and ZAct_manipulate_gripe
    | Act_move := not (Act_maint_cabin and Act_maint_track_left
                and Act_maint_track_right)
    | ZAct_move := Act_move$1 init false
    | u2 := Act_move and not ZAct_move
    | d2 := not Act_move and ZAct_move

    | a := u1
    | b := d1 and not u2
    | c := u1
    | d := d1 and u2
    | e := u2 and not d1
    | f := u1 and not d2
    | g := u1 and d2
    | h := not u1 and d2

    | (Start,ZStart) := m{true}(ZT2_active and h, a)
    | (T1_first,ZT1_first) :=
              m{false}((ZStart and a) or (ZT2_active and g), b or d)
    | (T1_idle,ZT1_idle) :=
              m{false}((ZT1_first and b), c or e)
    | (T2_active,ZT2_active) :=
              m{false}((ZT1_first and d) or (ZT1_idle and e), f or g)
    | (Error,ZError) :=
              m{false}((ZT1_idle and c) or (ZT2_active and f),
                       false when (^a))
    |)
    where
          ...
```

Table 11: SIGNAL code for an observer for the property: *exclude two successive manipulations of the grip without any cabin or base movement in between.*

7.3.3 Performances

We here present an overview of the computational results. Two methods have been used to perform the control:

- sub-system by sub-system: we performed the control first on each sub-system separately and then we composed the resulting controlled sub-systems before performing the control between the different sub-systems: Method (A). We distinguish between the cost of computation for only the properties local to each actuator, and the ones caring for objectives across subsystems (cross properties).
- on the whole system: we first combined all the processes in order to obtain a polynomial dynamical system that encodes the behavior of the whole excavation machine. We then apply the controller synthesis requests one by one on this resulting PDS : Method (B).

```
process obs2 =
     ( ? boolean Act_auto_arm;
         boolean Act_home_arm;
       ! boolean Error;
     )
   (| a := Act_auto_arm
    | b := (not Act_auto_arm) and Act_home_arm
    | c := (not Act_auto_arm) and (not Act_home_arm)

    | (Start,ZStart) := m{true}(ZInAct_auto_arm and c,a)
    | (InAct_auto_arm,ZInAct_auto_arm) :=
              m{false}(ZStart and a, b or c)
    | (Error,ZError) :=
              m{false}((ZInAct_auto_arm and b),
                       false when (^a))
    |)
    where
          ...
```

Table 12: SIGNAL code for an observer for the property: *between any two task where the arm moves, the default task should be activated.*

	Grip	Arm	Cabin	Base	Method (A)		Method (B)
X	4	8	6	14	32		"
Y(cont)	5(2)	9(4)	7(3)	16(7)	37(16)		"
Orb_uc	9	81	27	2187	43 046 721		"
					local properties	cross properties	"
Orb_c	4	32	12	192	294912	172032	
Time	0.0.2s	0.033s	0.14	2.84s	6.57	43.46s	50.34s
TDD Size	433	316	172	367	5235	10132	11147

In this table, X (resp. Y) corresponds to the number of state variables (resp. event variables). Orb_nc (resp. Orb_c) corresponds to the number of reachable states in the uncontrolled (resp. controlled) system. and TDD size corresponds to the number of node of the TDD encoding the transition relation of the system (the difference between 10132 and 11147 is due to the automatic variable reordering).

7.4 Simulation

At every phase in the above construction of the model and control, it is possible to obtain a simulator of the behaviors of the controlled system. This way, a user can observe how the behaviors change when adding one objective, and verify whether the constraints added correspond to the problem to be solved. A simulation consists of iterating, step by step, the following three operations:

1. *simulating the environment* is done through the uncontrollable inputs panel (see Fig. 18(a)), where one can enter the requests from the operator, and the events signalling termination of tasks.

2. *choosing among correct controls* is done through the controllable events panel (see Fig. 18(b)). Values ruled out by constraints are represented by non-selectable buttons. There can be possibly several allowed values, if the constraints do not completely determine the control from the inputs. In order to obtain a control function, i.e., an input-deterministic controller, the specification has to be strengthened, or optimization criteria (w.r.t. costs on states or events) have to be applied, or a random choice can be applied. Then, the control of the system is directed in a shared way by, partly, requests from the user, and partly, control by the automatism.

3. *the dynamical evolution* is observed with the task states display shown in Fig. 19. Changes in behaviors obtained after adding a control objective can show in, e.g., observing that a requested task goes into wait state when another, incompatible one is active, until termination.

(a) The uncontrollable inputs panel (b) The controllable inputs panel

Figure 18: Panels for interactive simulation.

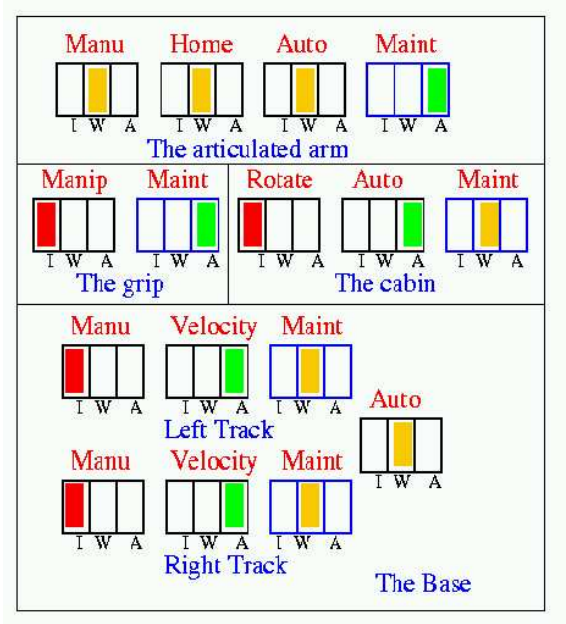

Figure 19: Task states display for simulation.

8 Conclusion and perspectives

8.1 Results

We have defined a framework for using discrete control synthesis in safe control systems programming, at the level of a set of control tasks. We have proposed a set of task and properties patterns, which correspond to concepts used by application designers. These patterns can be used for the construction of application models, and the application of dicrete control synthesis computation, without requiring deep knowledge of the underlying formal models and theory. In that sense, it proposes a user-friendly, automated use of this formal technique. The framework is systematic enough to design and develop an automated process on these bases. It could be integrated with a programming environment like ORCCAD. We have built up a practical experiment, using the SIGNAL/SIGALI environment. The obtained controllers are simulated in an interactive way, which can be interpreted as a form of teleoperation, where the control is shared between a human operator and an automatism, in a discrete space.

8.2 Perspectives

The relation with other approaches could be explored, e.g., techniques based on the supervisory control of Petri net-based models, modeling structures distinguishing supervision and control [19], or program generation in robotics, like reactive planning [67].

Extensions of the current work can be pursued in different directions. An interesting aspect concerns the enrichment of the task structure in order to represent multiple modes of activity, corresponding to different qualities (e.g., accurateness of models) and costs, or gradually degraded modes, e.g. in relation with fault tolerance [43]. Such an enriched model can be confronted with recent techniques in optimal discrete control synthesis [46] with which an automatic controller can be synthesized, always switching to the best solution. Also, it would be interesting to consider temporal or even hybrid extensions of transition systems and controller synthesis [7, 4]. The integration into a compilation process of model-checking is interesting, as a use of the dynamical semantics of the programs.

Another direction is the actual implementation, and application on experimental control or robotic systems. This should allow for the determination of more precise functionalities with usefulness w.r.t. the application domains. Other application domains, where multi-task systems are encountered, can also be explored, like systems on chip for example.

Acknowledgement

Acknowledgement is due to Prajna Prakash Parida (in the framework of the Indo-French IIT-INRIA exchange program) and Antonio Medina Rodriguez (in the framework of the CONACYT program of Mexico) for practical help, and to B. Espiau, D. Simon, A. Girault, F. Maraninchi, K. Altisen, amongst others, for constructive discussion.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *Int. J. of Robotics Research*, 17(4):315–337, April 1998.

[2] G. Alpan and M. A. Jafari. Synthesis of sequential controllers in the presence of conflicts and free choices. *IEEE Trans. on Robotics and Automation*, 14(3):488–492, June 1998.

[3] K. Altisen. Génération automatique d'ordonnancements pour systèmes temporisés. Rapport de DEA (MSc thesis), ENSIMAG, Grenoble, 1998. *(in French).*

[4] K. Altisen. *Application du principe de synthèse de contrôleur à l'ordonnancement de systèmes temps-réel.* PhD thesis, INPG, Grenoble, December 2001.

[5] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proceedings of the IEEE Real-Time Systems Symposium, RTSS'99*, 1999.

[6] T. Amagbegnon, P. Le Guernic, H. Marchand, and E. Rutten. Signal – the specification of a generic, verified production cell controller. *Formal Development of Reactive Systems – Case Study Production Cell,* volume 891 *of Lecture Notes in Computer Science,* Chapter VII, pages 115–129, January 1995.

[7] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers of linear systems. *Proceedings of the IEEE*, 88:1011–1025, 2000.

[8] F. Baccelli, B. Gaujal, and D. Simon. Analysis of preemptive periodic real time systems using the (max,plus) algebra with applications in robotics. *IEEE Trans. on Control Systems Technology*, 2002. *(to appear).*

[9] S. Balemi, G. J. Hoffmann, H. Wong-Toi, and G. F. Franklin. Supervisory control of a rapid thermal multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, July 1993.

[10] M. Barbeau, M. Frappier, F. Kabanza, and R. St-Denis. A supervisory control synthesis case-study: The antenna control system. In *Proceedings of the 35th Annual Allerton Conference on Communication, Control, and Computing, October 1997*, 1997. `www.dmi.usherb.ca/~kabanza/index.html`.

[11] M. Barbeau, F. Kabanza, and R. St-Denis. A method for the synthesis of controllers to handle safety, liveness, and real-time constraints. *IEEE Transactions on Automatic Control*, 43(11), 1998.

[12] Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[13] L. Besnard, P. Bournai, T. Gautier, N. Halbwachs, S. Nadjm-Tehrani, and A. Ressouche. Design of a multi-formalism application and distribution in a data-flow context: an example. In *Proceedings of the 12th International Symposium on Languages for Intensional Programming, ISLIP' 99, NCSR Demokritos, Athens, Greece, June 1999*, 1999.

[14] J-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The ORCCAD architecture. *Int. J. of Robotics Research*, 17(4):338–359, April 1998.

[15] B.A. Brandin. The real-time supervisory control of an experimental manufacturing cell. *IEEE Transactions on Robotics and Automation*, 12(1):1–14, 1996.

[16] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

[17] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM computing Surveys*, pages 293–318, September 1992.

[18] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.

[19] F. Charbonnier, H. Alla, and R. David. The supervised control of discrete event dynamic systems. *IEEE Trans. on Control Systems Technology*, 7(2):175–187, March 1999.

[20] E. Coste-Manière and B. Espiau. Special issue on integrated architecture for robot control and programming. *Int. J. of Robotics Research*, 17(4), April 1998.

[21] B. Dutertre. *Spécification et preuve de systèmes dynamiques.* PhD thesis, Université de Rennes I, IFSIC, December 1992.

[22] H. Fargier, M.Jourdan, N. Layaïda, and T. Vidal. Using temporal constraints networks to manage temporal scenario of multimedia documents. In *Proceedings of the ECAI'98 Workshop on Spatial and Temporal Reasoning, Brighton, England, August 1998*, 1998.

[23] J.L. Fleureau, L. Nana Tchamnda, L. Marcé, and L. Abalain. Remote-controlled vehicle using pilot language. In *Proceedings of ANS'99, American Nuclear Society, Pittsburgh, Pennsylvania*, April 1999.

[24] Dominique Francisci. Utilisation de méthodes formelles pour analyser des applications de robotique avec un haut niveau d'abstraction. Rapport de stage de DEA, Université de Nice, ESSI, June 2000. *(in French)*.

[25] S. Gaubert. *Systèmes Dynamiques à événements discrets.* ENSMP, Orsay, 1996. lecture notes (*in French*).

[26] J. Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems.* PhD thesis, Linköping University, 1997.

[27] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[28] N. Halbwachs. Synchronous programming of reactive systems – a tutorial and commented bibliography. In *Proc. of the Int. Conf. on Computer-Aided Verification, CAV'98,* Vancouver, Canada. Springer-Verlag, 1998. LNCS nr. 1427.

[29] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[30] L.E. Holloway, B.H. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Application*, 7:151–190, 1997.

[31] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model checking for managers. In *Proceedings of the 6th International SPIN Workshop on Practical Aspects of Model Checking, Toulouse, France, September 1999*, number 1680 in Lecture Notes in Computer Science (LNCS), pages 92–107. Springer Verlag, 1999.

[32] M. Jeng and X. Xie. ERCN-merged nets and their analysis using siphons. *IEEE Trans. on Robotics and Automation*, 15(4):692–703, 1999.

[33] M.D. Jeng and X.L. Xie. Modeling and analysis of semiconductor manufacturing systems with degraded behavior using petri nets and siphons. In *Proc. IEEE Int. Conf. on Robotics and Automation, ICRA'2001,* Seoul, Korea, may, 2001.

[34] M. Jourdan. Integrating formal verification methods of quantitative real-time properties into a development environment for robot controllers. Research Report 2540, INRIA, April 1995. (`www.inria.fr`).

[35] A. Kountouris and C. Wolinski. False path analysis based on a hierarchical control representation. In *Proceedings of ISSS'98*, Hsinchu, Taiwan, R.O.C., December 1998.

[36] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[37] M. Le Borgne. *Systèmes dynamiques sur des corps finis.* PhD thesis, Université de Rennes I, September 1993.

[38] M. Le Borgne, A. Benveniste, and P. Le Guernic. Polynomial dynamical systems over finite fields. In *Algebraic Computing in Control*, volume 165, pages 212–222. LNCIS, G. Jacob et F. Lamnabhi-lagarrigue, March 1991.

[39] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE, Special issue on the synchronous approach to real-time programming*, 79(9):1321–1336, September 1991.

[40] L. Libeaut and J.J. Loiseau. Commande de SED – théorie des langages commandables. In *Actes de la 26ème école de printemps d'informatique théorique : Algèbres Max-Plus et applications en informatique et automatique, INRIA-LIAFA-IRCyN, mai 1998.*, 1998. (*in French*).

[41] O. Maler, A.Pnueli, and J.Sifakis. On the synthesis of discrete controllers for timed systems. In *Proceedings of STACS'95*, volume 900 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, 1995.

[42] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E. W. Mayr and C. Puech, editors, *Proceedings STACS'95*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242. Springer-Verlag, 1995.

[43] F. Maraninchi, Y. Rémond, and E. Rutten. Effective programming language support for discrete-continuous mode-switching control systems. In *Proc. of the 40th IEEE Conf. on Decision and Control, CDC'01,* Orlando, Florida, dec, 2001.

[44] Florence Maraninchi and Yann Rémond. Mode-automata: about modes and states for reactive systems. In C. Hankin, editor, *Programming Languages and Systems, Proceedings of the 7th Eropean Symposium on Programming, ESOP'98, Lisbon, Portugal, march–april 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 185–199. Springer-Verlag, March 1998.

[45] E. Marchand, E. Rutten, H. Marchand, and F. Chaumette. Specifying and verifying active vision-based robotic systems with the Signal environment. *International Journal of Robotics Research*, 17(4):418–432, April 1998.

[46] H. Marchand, O. Boivineau, and S. Lafortune. Optimal control of discrete event systems under partial observation. In *Proc. of the 40th IEEE Conf. on Decision and Control, CDC'01,* Orlando, Florida, December 2001.

[47] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamical System: Theory and Applications*, 10(4):325–346, October 2000.

[48] H. Marchand, É. Rutten, M. Le Borgne, and M. Samaan. Formal verification of programs specified with SIGNAL : Application to a power transformer station controller. *Science of Computer Programming*, 41(1):85–104, 2001.

[49] H. Marchand and M. Samaan. Incremental design of a power transformer station controller using controller synthesis methodology. *IEEE Transaction on Software Engineering*, 26(8):729–741, August 2000.

[50] Hervé Marchand, Prajna Prakash Parida, and Éric Rutten. Discrete control synthesis made easy: a user's manual for SIGALI. Technical report, INRIA, 2002. *(to appear).*

[51] H. Melcher and K. Winkelman. Controller synthesis for the 'production cell' case study. In *Proceedings of the 2nd workshop on Formal Methods on Software Practice, FMSP'98, New York, march 4-5.* ACM Press, 1998.

[52] D. Musliner, R. Goldman, and M. Pelican. Using model-checking to guarantee safety in automatically-synthesized real-time controllers. In *Proceedings of the 2000 IEEE Int. Conf. on Robotics and Automation, San Francisco, California*, April 2000.

[53] C. Ndjab and J. Zaytoon. Synthèse d'une implantation optimale de la commande à partir du grafcet. In *Actes du 2ième Congrès Modélisation des Systèmes Réactifs, MSR'99, Cachan, France, 24-25 Mars 1999*, pages 193–202, 1999.

[54] J.-C. Paoletti, E. Rutten, and L. Marcé. A language for modelling and monitoring in telerobotics. In *Proceedings of the 9^{th} European Annual Conference on Human Decision making and Manual Control,* Ispra, Varese, Italy, September 10–12, 1990.

[55] P. P. Parida. A case study in discrete controller synthesis: from specification to graphical simulation. Training report, dept. of computer science and engineering, I.I.T. Kahragpur, July 2001.

[56] S. Pinchinat. Sigali *vs* μcalculus. Personnal communication, 1996.

[57] S. Pinchinat and H. Marchand. Symbolic abstractions of automata. In *Proc of 5th Workshop on Discrete Event Systems, WODES 2000*, pages 39–48, Ghent, Belgium, August 2000.

[58] L.E. Pinzon, H.-M. Hanisch, M.A. Jafari, and T. Boucher. A comparative study of synthesis methods for discrete event controllers. *Formal Methods in System Design*, 15:123–167, 1999.

[59] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE, Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[60] E. Le Rest, J.L. Fleureau, and L. Marcé. Semantics and implementation of a language for telerobotics. In *Proc. of IROS'97,IEEE/RSJ,* Grenoble, France, September 1997.

[61] E. Rutten. A framework for using discrete control synthesis in safe robotic programming and teleoperation. In *Proceedings of the IEEE International Conference on Robotics and Automation, ICRA'2001,* may 21–26, Seoul, Korea, pages 4104–4109, 2001.

[62] Eric Rutten, Eric Marchand, and François Chaumette. An experiment with reactive data-flow tasking in active robot vision. *Software – Practice & Experience*, 27(5):599–621, May 1997.

[63] Eric Rutten and Florent Martinez. Signalgti: implementing task preemption and time intervals in the synchronous data flow language signal. In *Proceedings of the 7th Euromicro Workshop on Real Time Systems, ERTS'95,* Odense, Denmark, June 14 - 16, 1995, pages 176–183. (IEEE Publ.), 1995.

[64] R. Sengupta and S. Lafortune. An optimal control theory for discrete event systems. *SIAM Journal on Control and Optimization*, 36(2), march 1998.

[65] D. Simon, M. Personnaz, and R. Horaud. TELEDIMOS: Telepresence simulation platform for civil work machines : real-time simulation and 3d vision reconstruction. In *Proceedings of the IARP Workshop on Advances in Robotics for Mining and Underground Applications, Brisbane, Australia, October 2-4 2000*, 2000.

[66] L. Nana Tchamnda, J.L. Fleureau, and L. Marcé. A control system for pilot : software architecture and implemantation issues. In *Proceedings of ANS'01, ANS 9th International Topical Meeting on Robotics and Remote Systems, Seattle, Washington*, March 2001.

[67] S. Thiébaux and J. Hertzberg. A semi-reactive planner based on a possible models action formalization. In *Proceedings of the 1st International Conference on AI Planning Systems, AIPS'92, College Park (MD), June 1992*, pages 228–235. Morgan Kaufmann, 1992.

[68] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and timed systems. In *Proceedings of the World Congress on Formal Methods (FM'99), Toulouse, France, September 20-24, 1999*, number 1709 in Lecture Notes in Computer Science (LNCS). Springer Verlag, 1999.

[69] N. Turro, E. Coste-Manière, and O. Khatib. A portable programming framework. In *Experimental Robotics VI*, number 250 in LNCIS. Springer Verlag, 2000.

[70] UMDES-LIB. DES group, University of Michigan, Ann Arbor, MI, USA, available at http://www.eecs.umich.edu/umdes/projects.html#lib.

[71] W. M. Wonham. Notes on control of discrete-event systems. Technical Report ECE 1636F/1637S, Department of Electrical and Computer EngineeringUnivertsity of Toronto, 1999.

[72] Z.H. Zhang and W.M. Wonham. Stct: An efficient algorithm for supervisory control design. In *Symposium on Supervisory Control of Discrete Event Systems (SCODES2001)*, Paris (France), July 2001.

Contents

List of Figures

List of Tables

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopˆole de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399